Módulo 4: El CRUD y la Lógica de Negocio (Manipulación de Datos)

En este módulo, implementaremos la funcionalidad completa de un servicio REST: **CRUD (Create, Read, Update, Delete)**.
Abandonamos nuestra base de datos de simulación (db_items = {}) y la reemplazamos por un archivo database.json. Esto nos obliga a gestionar la carga, consulta y guardado de datos, dándonos el escenario perfecto para usar técnicas avanzadas de manipulación de listas y diccionarios en Python.

4.1. Configuración de la "Base de Datos" JSON

Vamos a crear un archivo database.json en la raíz de nuestro proyecto.
Para interactuar con él, modificaremos nuestro main.py:

1. **Cargaremos** el JSON en una variable (una list[dict]) al iniciar la API.
2. Crearemos una **función *helper*** para *guardar* esta variable de vuelta en el archivo database.json cada vez que hagamos un cambio (en POST, PUT, DELETE).

4.2. Manejo de Errores: HTTPException

¿Qué pasa si un cliente pide GET /items/99 y el ítem 99 no existe?

- **Mal:** Devolver {"error": "Item not found"} con un código 200 OK. El código 200 miente, ya que la operación no fue exitosa.
- **Mal (Peor):** Dejar que la API crashee (ej. KeyError) y devuelva un 500 Internal Server Error.
- **Bien (RESTful):** Devolver un código de estado **404 Not Found** con un mensaje claro.

FastAPI nos da la clase HTTPException para esto. Cuando queramos detener la ejecución y devolver un error HTTP, haremos:

```
from fastapi import FastAPI, HTTPException

# ...
raise HTTPException(status_code=404, detail="Item no encontrado")
```

FastAPI se encargará de generar la respuesta JSON de error con el código 404.

4.3. Implementando el CRUD

Usaremos las técnicas que solicitaste (lambda, list comprehensions) para que el código sea idiomático y eficiente.

1. READ (Lectura)

A. GET /items/ (Leer todos)

Este *endpoint* devolverá la lista completa. Le añadiremos **parámetros de consulta (Query Parameters)** para filtrar, por ejemplo, por precio.

```
# main.py
from fastapi import Query

@app.get("/items/", response_model=list[Item])
def read_items(max_price: float | None = Query(None, gt=0)):
    """
    Lee todos los ítems de la base de datos.

    - **max_price** (opcional): Filtra ítems con un precio menor o igual.
    """
    if max_price:
        # ¡List Comprehension al rescate!
        # Filtramos la BBDD en memoria.
        filtered_items = [item for item in db if item["price"] <= max_price]
        return filtered_items

    return db
```

- Query(None, gt=0): Usamos Query para añadir validación extra al parámetro (en este caso, gt=0 significa "greater than 0").
- [item for item in db if ...]: Esta es la forma *Pythonica* de filtrar una colección. Es más rápida que un bucle for que añade (.append()) a una lista nueva.

B. GET /items/{item_id} (Leer uno)

Aquí buscaremos un ítem específico. Usaremos lambda con filter.

```
# main.py

@app.get("/items/{item_id}", response_model=Item)
def read_item(item_id: int):
    # Usamos filter() y una lambda. filter() devuelve un iterador.
    # next(..., None) toma el primer resultado o devuelve None si no hay.
    item = next(filter(lambda x: x["id"] == item_id, db), None)

    if item is None:
        raise HTTPException(status_code=404, detail="Item no encontrado")

    return item
```

2. CREATE (Creación)

POST /items/

Aquí, debemos:
1. Validar el ItemCreate (lo hace Pydantic).
2. Generar un nuevo id.
3. Añadir el ítem a nuestra lista db.
4. **Guardar** la lista db en database.json.

```
# main.py

@app.post("/items/", response_model=Item, status_code=201)
def create_item(item: ItemCreate):
    """
    Crea un nuevo ítem.
    """
    # Lógica para generar un nuevo ID
    # Usamos max() con una list comprehension y un key lambda
    # (o un generador)
    last_id = max((i["id"] for i in db), default=0)
    new_id = last_id + 1

    new_item_data = item.model_dump()
    new_item_data["id"] = new_id

    # "Guardamos" en la BBDD en memoria
    db.append(new_item_data)

    # Guardamos en el archivo JSON
    save_db()

    return new_item_data
```

- status_code=201: Por semántica REST, una creación exitosa debe devolver 201 Created.

3. UPDATE (Actualización)

PUT /items/{item_id}

PUT es semánticamente un **reemplazo total**. El cliente debe enviar *todos* los campos del ítem (excepto el id).

```
# main.py

@app.put("/items/{item_id}", response_model=Item)
```

```
def update_item(item_id: int, item_update: ItemCreate):
    """
    Actualiza (reemplaza) un ítem por su ID.
    """
    # Buscamos el índice del ítem en la lista
    # Usamos enumerate() para obtener (índice, ítem)
    index = next((i for i, item in enumerate(db) if item["id"] == item_id), None)

    if index is None:
        raise HTTPException(status_code=404, detail="Item no encontrado")

    # Creamos el nuevo objeto (preservando el ID original)
    updated_item_data = item_update.model_dump()
    updated_item_data["id"] = item_id

    # Reemplazamos el ítem en la lista
    db[index] = updated_item_data

    save_db()
    return updated_item_data
```

4. DELETE (Borrado)

DELETE /items/{item_id}

Aquí es donde una list comprehension brilla. La forma más eficiente de "eliminar" un ítem de una lista no es list.remove(), sino **reconstruir la lista *sin* ese ítem**.

```
# main.py

@app.delete("/items/{item_id}", status_code=200)
def delete_item(item_id: int):
    """
    Elimina un ítem por su ID.
    """
    global db # Necesario para reasignar la variable global

    # Primero, comprobamos si existe
    item_exists = any(item["id"] == item_id for item in db)
    if not item_exists:
        raise HTTPException(status_code=44, detail="Item no encontrado")

    # List comprehension: reconstruimos la lista
    # incluyendosolo los IDs que NO coinciden.
```

```
    db = [item for item in db if item["id"] != item_id]

    save_db()

    return {"detail": "Item eliminado correctamente", "id": item_id}
```

- global db: Necesario porque no estamos *modificando* la lista (.append), sino *reasignando* la variable db a una lista completamente nueva.

Con esto, tenemos un servicio RESTful completo, robusto (con manejo de errores 404) y con lógica de negocio clara y eficiente.