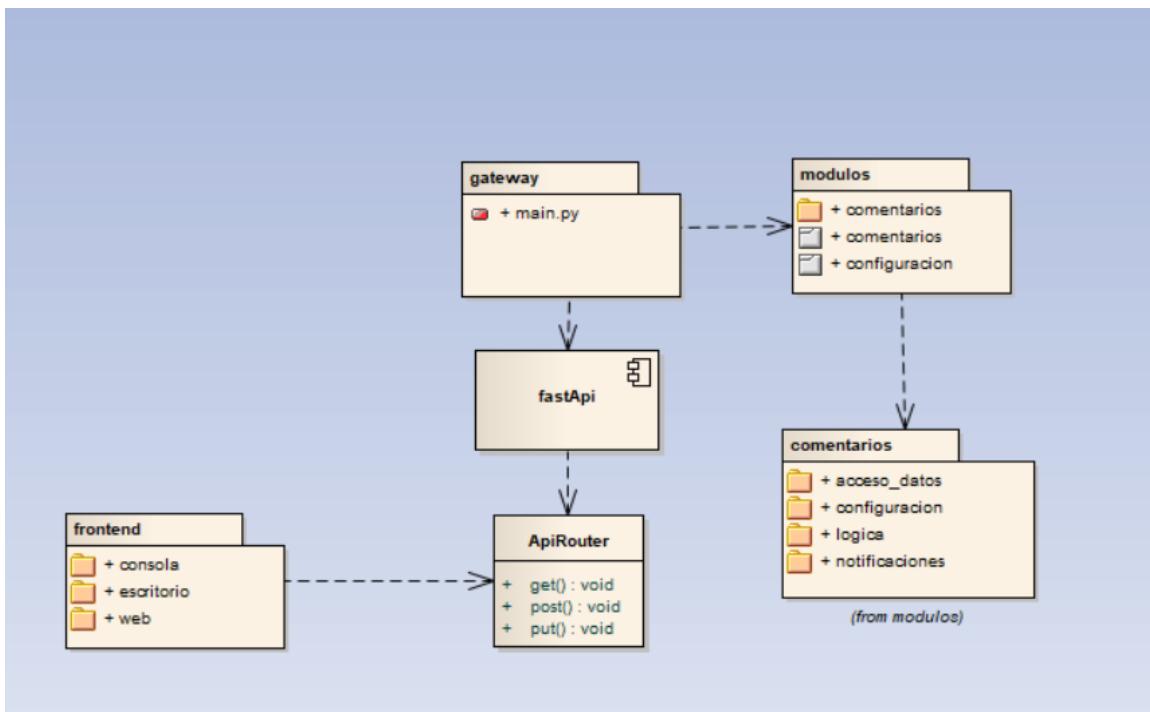


## Guía Proyecto con Arquitectura de Microservicios



## 0. Principios de Diseño

---



## Inyección de Dependencias (Dependency Injection)

Es un **principio de diseño** que consiste en **no crear directamente los objetos que una clase necesita**, sino **recibirlos desde el exterior**. Así, una clase **no se encarga de construir sus propias dependencias**, sino que estas le son "inyectadas" por otro componente.

### ✖ ¿Por qué es útil?

- Favorece el **desacoplamiento** entre clases.
- Facilita la **prueba unitaria** (se pueden simular dependencias).
- Permite cambiar implementaciones sin modificar la clase que las usa.

### ✖ Ejemplo sin inyección de dependencias

```
class ServicioCorreo:  
    def enviar(self, mensaje):  
        print(f"Enviando correo: {mensaje}")  
  
class Notificador:  
    def __init__(self):  
        self.servicio = ServicioCorreo() # dependencia *creada* internamente  
  
    def notificar(self, texto):  
        self.servicio.enviar(texto)  
  
# Uso  
n = Notificador()  
n.notificar("Tienes una nueva alerta.")
```

✖ **Problema:** Notificador depende directamente de ServicioCorreo. Si quiero cambiar a ServicioSMS o simularlo en una prueba, tengo que modificar el código de la clase.

---

### ✓ Ejemplo con inyección de dependencias

```

class ServicioCorreo:
    def enviar(self, mensaje):
        print(f"Enviando correo: {mensaje}")

class Notificador:
    def __init__(self, servicio):
        self.servicio = servicio # dependencia *inyectada*

    def notificar(self, texto):
        self.servicio.enviar(texto)

# Uso
correo = ServicioCorreo()
n = Notificador(correo)
n.notificar("Tienes una nueva alerta.")

```

### Ventajas:

- Notificador puede funcionar con **cualquier clase que tenga el método enviar()**.
- En pruebas, podrías inyectar un **servicio simulado (mock)**.
- El cambio de implementación no afecta a la lógica de Notificador.

**Sin inyección:** la clase crea sus propias dependencias → fuerte acoplamiento.

**Con inyección:** las dependencias se pasan desde fuera → bajo acoplamiento, más flexible y testable.

---

## Inversión de Dependencias (Dependency Inversion Principle – DIP)

La **Inversión de Dependencias** es uno de los cinco principios SOLID del diseño orientado a objetos.

### Enunciado del principio:

"Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones."

---

 **Intuitivamente significa:**

- En lugar de que una clase principal dependa directamente de implementaciones concretas (módulos de bajo nivel), debe depender de interfaces o abstracciones.
  - Las clases concretas (implementaciones) deben **adaptarse** a esas abstracciones, no al revés.
- 

 **Ejemplo sin inversión de dependencias**

```
class MotorGasolina:  
    def encender(self):  
        print("Motor de gasolina encendido.")  
  
class Auto:  
    def __init__(self):  
        self.motor = MotorGasolina() # Auto depende directamente del motor concreto  
  
    def arrancar(self):  
        self.motor.encender()
```

 **Problema:** si mañana quieres usar un motor eléctrico, tienes que reescribir la clase Auto. Está fuertemente acoplada a MotorGasolina.

---

 **Ejemplo con inversión de dependencias**

```

# Abstracción
class IMotor:
    def encender(self):
        pass

# Implementación concreta 1
class MotorGasolina(IMotor):
    def encender(self):
        print("Motor de gasolina encendido.")

# Implementación concreta 2
class MotorElectrico(IMotor):
    def encender(self):
        print("Motor eléctrico encendido.")

# Módulo de alto nivel
class Auto:
    def __init__(self, motor: IMotor):
        self.motor = motor # depende de una abstracción

    def arrancar(self):
        self.motor.encender()

```



### Ventajas:

- Auto no necesita saber qué tipo de motor se usa. Solo espera algo que implemente IMotor.
- Puedes intercambiar motores (gasolina, eléctrico, híbrido) **sin tocar la clase Auto**.
- Ideal para pruebas unitarias, mantenimiento y escalabilidad.

---

### Diferencias clave

## Sin DIP

La clase depende de otra concreta

Alto acoplamiento

Diffícil de modificar o extender

Rígido ante cambios

## Con DIP

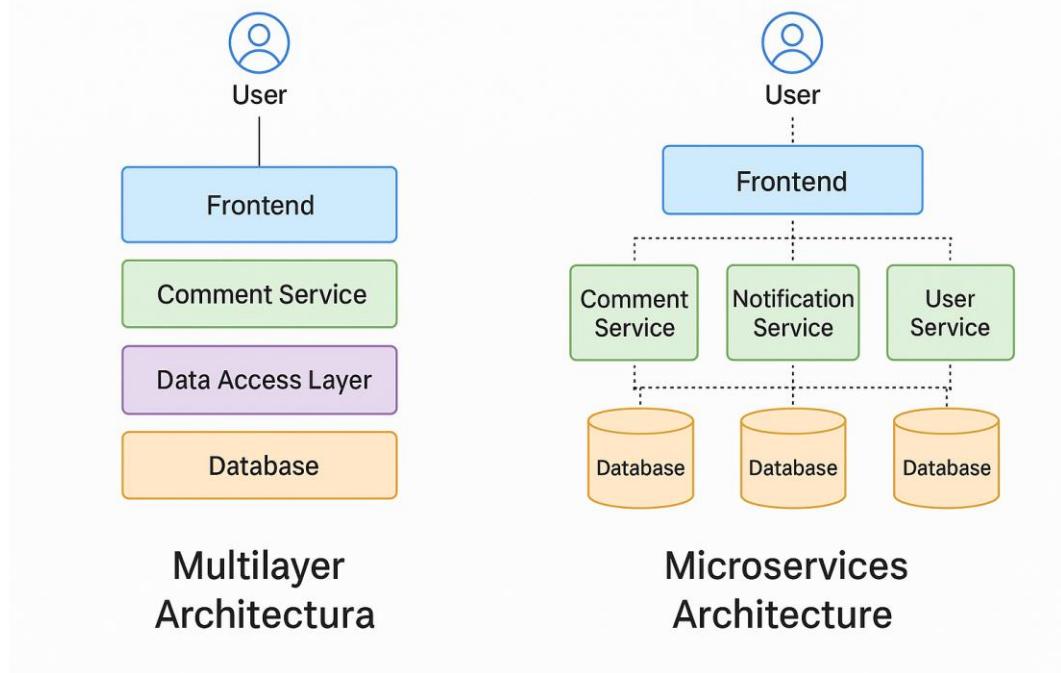
La clase depende de una abstracción

Bajo acoplamiento

Fácil de cambiar y escalar

Flexible ante nuevas implementaciones

## Arquitectura Multicapa vs Arquitectura Microservicios



### Arquitectura Multicapa (Proyecto: Lab-Multicapa-Python)

#### Separación lógica en capas

El sistema se organiza en capas: presentación, lógica de negocio, acceso a datos y configuración.

Cada capa **importa** explícitamente a la inferior (por ejemplo, la lógica importa los DAOs desde la capa de acceso a datos).

#### Acoplamiento por dependencias directas

Las dependencias se resuelven mediante importaciones dentro del mismo proceso.

El flujo de ejecución se da verticalmente: UI → lógica → DAO → base de datos.

#### Ejecución centralizada

**Todo el sistema se ejecuta como un único programa.**

**El control está centralizado, y no hay componentes distribuidos en red.**

#### Ventajas

- Facilidad de desarrollo inicial.
- Buena organización para proyectos monolíticos pequeños o medianos.

#### Limitaciones

- Dificultad para escalar horizontalmente.
- Cambiar o actualizar una parte requiere redeploy de todo el sistema.
- Las capas están acopladas a nivel de código (importaciones), dificultando el desacoplamiento real.

---

### ✳ Arquitectura de Microservicios (Proyecto: ProyectoGatewayEstructuraFinal)

#### Descomposición en servicios independientes

- Cada funcionalidad (como comentarios o notificaciones) está implementada como un microservicio autónomo.
- No existe importación entre microservicios; la comunicación se realiza vía HTTP a través del API Gateway.

#### Desacoplamiento total

- Los servicios están completamente desacoplados: cada uno tiene su propia estructura de carpetas, DAO, lógica, configuración e incluso pueden ejecutarse en distintos entornos o lenguajes.
- La coordinación se hace por integración (comunicación), no por integración de código.

#### Ejecución distribuida

- Cada microservicio se ejecuta de forma independiente, en su propio proceso y, potencialmente, en su propio contenedor o servidor.
- El archivo config.json y la función obtener\_fabrica() permiten decidir dinámicamente el motor de base de datos por microservicio.

#### Ventajas

- Escalabilidad horizontal: cada servicio puede replicarse o escalarse de manera independiente.
- Mantenibilidad: es posible modificar o desplegar un microservicio sin afectar al resto del sistema.
- Tolerancia a fallos y autonomía tecnológica.

#### Limitaciones

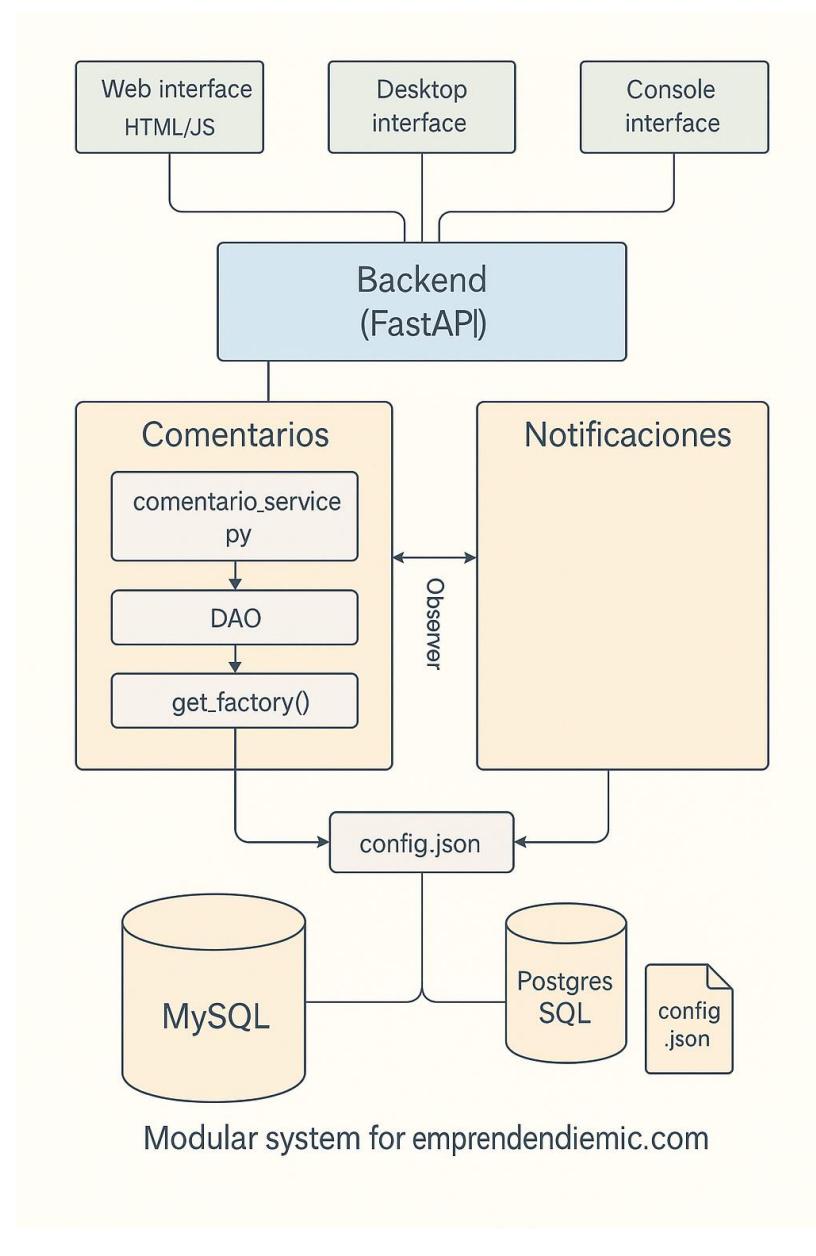
- Mayor complejidad en el despliegue y la orquestación.
- Necesidad de herramientas adicionales para registro, monitoreo, y balanceo de carga.
- Latencia y dependencia de red entre componentes.

---

### ⌚ Diferencia clave entre ambos modelos

- En la arquitectura **multicapa**, los módulos de alto nivel **importan directamente** los módulos de bajo nivel (por ejemplo, from acceso\_datos import comentario\_dao).
- En la arquitectura de **microservicios**, **no se realiza importación de capas entre módulos**. Cada servicio se comunica de manera **externa e independiente**, típicamente vía HTTP, favoreciendo el **desacoplamiento fuerte** y la autonomía de despliegue.

## 1. Descripción del Proyecto final (Arquitectura Microservicio)



### ✿ Arquitectura General del Proyecto

#### 1. Enfoque basado en microservicios

- Cada funcionalidad del sistema se implementa como un servicio independiente, autónomo y comunicable vía HTTP.
- Esto permite escalar, desplegar y mantener cada componente por separado.

#### 2. Componentes principales

- gateway/: actúa como **API Gateway**, centralizando el enrutamiento de peticiones hacia los microservicios.
- modulos/comentarios/: microservicio dedicado a la gestión de comentarios.
- frontend/: contiene las tres interfaces de usuario:
  - consola/
  - escritorio/
  - web/

---

#### Estructura de Carpetas

```
ProyectoFinal/
└── gateway/
    └── main.py
├── modulos/
    └── comentarios/
        ├── acceso_datos/
        ├── configuracion/
        ├── logica/
        └── notificaciones/
└── frontend/
    ├── consola/
    ├── escritorio/
    └── web/
└── requirements.txt
```

---

#### Detalle de Carpetas Clave

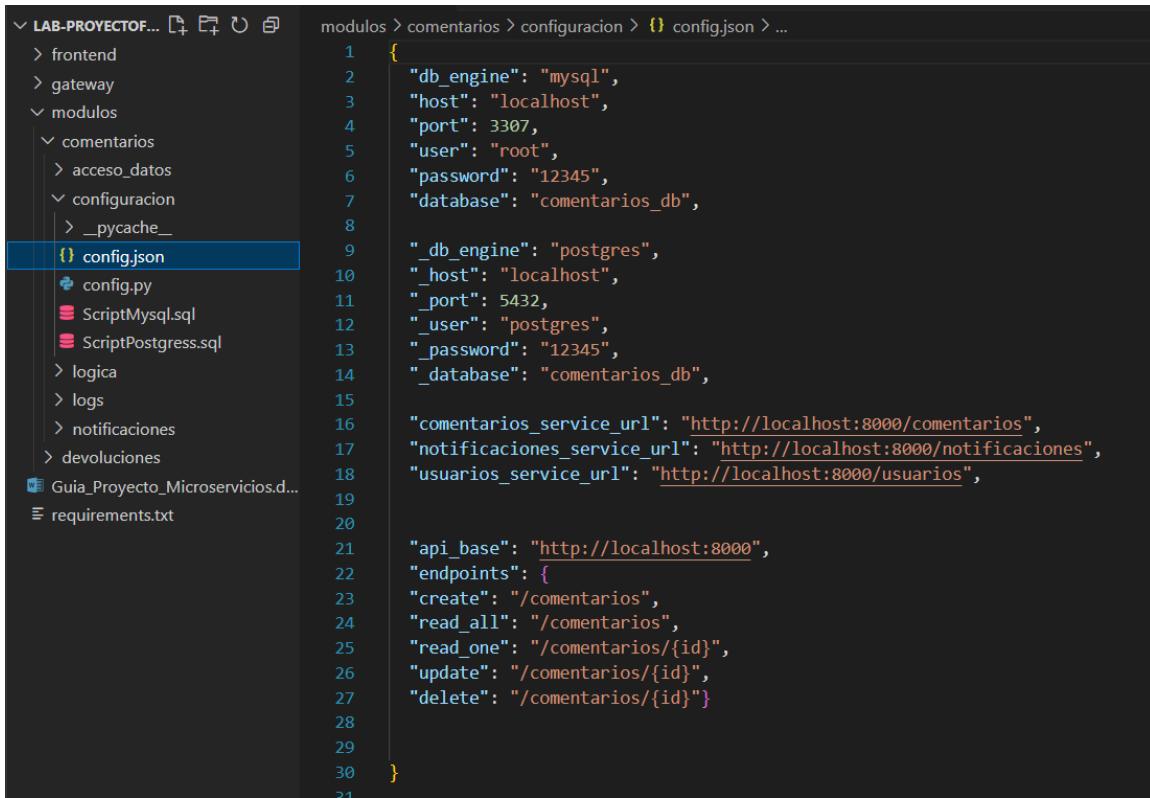
- **gateway/**
  - Contiene el punto de entrada principal (main.py).
  - Implementa el patrón **API Gateway**, actuando como único canal de acceso para los clientes.
  - Centraliza funciones como autenticación, enrutamiento y monitoreo.
- **modulos/comentarios/ (Microservicio)**
  - acceso\_datos/: contiene la implementación de los patrones **DAO** y **DTO**.
  - configuracion/: maneja los archivos config.json y scripts SQL.
  - logica/: expone la API REST usando **FastAPI**.
  - notificaciones/: implementa el patrón **Observer** para emitir alertas o eventos ante comentarios críticos.
- **frontend/**
  - consola/: interfaz basada en línea de comandos.
  - escritorio/: construida con **Tkinter**, ejecutable en local.
  - web/: interfaz ligera basada en HTML, CSS y JavaScript.

---

#### Extensibilidad del sistema

- La carpeta modulos/ está diseñada con principios de cohesión y extensibilidad.
  - Es posible agregar nuevos microservicios como:
    - devoluciones/
    - facturacion/
    - usuarios/
  - Cada nuevo módulo puede incluir su propio acceso\_datos, logica, configuracion y otros subcomponentes, sin afectar la estructura existente.
-

## modulos\comentarios\configuración



```
1  {
2      "db_engine": "mysql",
3      "host": "localhost",
4      "port": 3307,
5      "user": "root",
6      "password": "12345",
7      "database": "comentarios_db",
8
9      "_db_engine": "postgres",
10     "_host": "localhost",
11     "_port": 5432,
12     "_user": "postgres",
13     "_password": "12345",
14     "_database": "comentarios_db",
15
16     "comentarios_service_url": "http://localhost:8000/comentarios",
17     "notificaciones_service_url": "http://localhost:8000/notificaciones",
18     "usuarios_service_url": "http://localhost:8000/usuarios",
19
20     "api_base": "http://localhost:8000",
21     "endpoints": {
22         "create": "/comentarios",
23         "read_all": "/comentarios",
24         "read_one": "/comentarios/{id}",
25         "update": "/comentarios/{id}",
26         "delete": "/comentarios/{id}"
27     }
28
29
30 }
31
```

### Parámetros de conexión para MySQL (db\_engine, host, port, user, password, database)

- Define la configuración necesaria para establecer una conexión con un motor de base de datos MySQL.
- Estos valores son usados por el componente DAO del microservicio para inicializar conexiones dinámicamente.
- Promueve la externalización de parámetros críticos, evitando su codificación directa en el programa.

### Parámetros de conexión para PostgreSQL (\_db\_engine, \_host, \_port, etc.)

- Proporciona una configuración alternativa para entornos que requieren PostgreSQL.
- La existencia de ambas configuraciones permite al sistema cambiar de motor de base de datos modificando únicamente el archivo config.json, sin necesidad de alterar el código fuente.
- Refleja un enfoque multibase de datos basado en el principio de configuración por entorno.

### URLs de microservicios (\*\_service\_url)

- comentarios\_service\_url, notificaciones\_service\_url y usuarios\_service\_url especifican las rutas base para consumir los distintos microservicios que conforman la arquitectura distribuida.

- Estas URLs son utilizadas por el gateway o por otros servicios para realizar peticiones HTTP internas, promoviendo la interoperabilidad y la independencia entre módulos.

### Ruta base del API (api\_base)

- Define la URL raíz sobre la cual se construyen los endpoints específicos.
- Permite que toda la lógica de interacción con la API pueda modificarse a otro servidor o puerto simplemente ajustando este valor en la configuración.

### Estructura de endpoints (endpoints)

- El objeto endpoints mapea las acciones CRUD del microservicio de comentarios:
  - "create": ruta para registrar nuevos comentarios (POST).
  - "read\_all": ruta para listar todos los comentarios (GET).
  - "read\_one": ruta con parámetro {id} para obtener un comentario específico (GET).
  - "update": ruta con {id} para modificar un comentario (PUT).
  - "delete": ruta con {id} para eliminar un comentario (DELETE).
- Estos endpoints son consumidos por las interfaces (web, escritorio y consola) y facilitan la reutilización centralizada de rutas.

### Buenas prácticas implementadas

- **Separación de configuración del código:** el archivo permite modificar detalles operativos sin alterar el software.
- **Flexibilidad y portabilidad:** ideal para despliegues en distintos entornos (desarrollo, pruebas, producción).
- **Compatibilidad con arquitectura de microservicios:** define rutas y servicios de manera desacoplada, facilitando el escalado y mantenimiento del sistema.

The screenshot shows a file explorer on the left and a code editor on the right. The file explorer displays a project structure with the following hierarchy:

```

LAB-PROYECTOFO...
  > frontend
  > gateway
  < modulos
    < comentarios
      > acceso_datos
    < configuracion
      > __pycache__
        config.json
      config.py
      ScriptMySQL.sql
      ScriptPostgress.sql
    > logica
    > logs
    > notificaciones
    > devoluciones
  Guia_Proyecto_Microservicios.d...
  requirements.txt

```

The code editor shows the content of `config.py`:

```

# configuracion/config.py
import json
def cargar_configuracion(path="modulos/comentarios/configuracion/config.json"):
    with open(path) as f:
        return json.load(f)

```

### archivo config.py – Carga de configuración

#### 1. Importación del módulo json

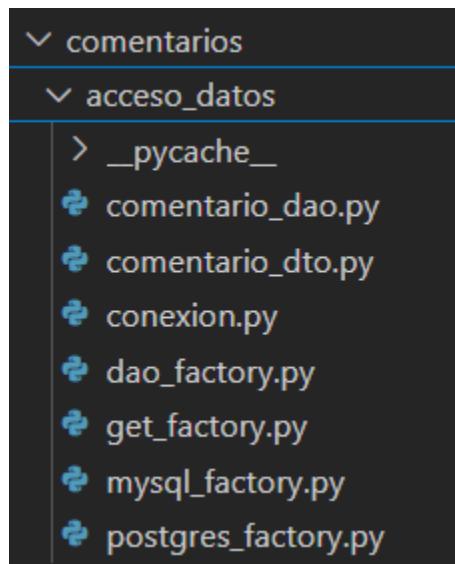
- El archivo comienza importando la biblioteca estándar json de Python.

- Esto permite manipular archivos en formato JSON, ampliamente utilizado para representar configuraciones y estructuras de datos en aplicaciones modernas.
- 2. Definición de la función cargar\_configuracion()**
- Esta función encapsula la lógica necesaria para leer y cargar dinámicamente la configuración del sistema desde el archivo config.json.
- 3. Parámetro path con valor por defecto**
- La función recibe como argumento un parámetro path, que apunta por defecto a: "modulos/comentarios/configuracion/config.json"
  - Esto permite mantener la flexibilidad: si se desea cargar otro archivo de configuración en otro entorno (por ejemplo, pruebas o producción), se puede cambiar el valor del parámetro al invocar la función.
- 4. Bloque with open(...) as f:**
- Se utiliza la estructura with, que es una buena práctica para el manejo de archivos en Python.
  - Garantiza que el archivo se cierre automáticamente luego de ser procesado, evitando posibles errores de manejo de recursos.
- 5. Devolución de los datos como estructura Python**
- La función retorna directamente el resultado de json.load(f), que convierte el contenido del archivo JSON en un diccionario de Python.
  - Este diccionario puede ser utilizado por cualquier componente del sistema para acceder de forma estructurada a parámetros como rutas, credenciales o endpoints.

### Principios aplicados

- **Separación de configuración del código:** evita la codificación directa de parámetros en los scripts principales.
- **Modularidad:** al encapsular esta funcionalidad en una función reutilizable, se mejora la mantenibilidad del sistema.
- **Adaptabilidad:** el uso de un parámetro con valor por defecto permite extender su uso a distintos entornos sin modificar el cuerpo de la función.

## modulos\comentarios\acceso\_datos



## Capa de Acceso a Datos del Módulo comentarios (acceso\_datos/)

La carpeta acceso\_datos representa la **capa de acceso a datos** del módulo comentarios, y su propósito fundamental es encapsular toda la lógica relacionada con la persistencia y recuperación de información desde el sistema de almacenamiento (base de datos). Esta capa está diseñada de forma modular y extensible, aplicando principios de diseño orientado a objetos y patrones arquitectónicos como **DAO**, **DTO** y **Abstract Factory**, lo cual permite desacoplar la lógica de negocio de los mecanismos específicos de persistencia. A continuación, se describen los principales componentes que la conforman:

---

### Descripción de los componentes

#### `comentario_dao.py`

- Define una interfaz base para operaciones CRUD sobre la entidad Comentario.
- Implementa el patrón **DAO (Data Access Object)**, proporcionando un contrato uniforme para las operaciones de persistencia, sin acoplarse a un motor de base de datos específico.
- Mejora la mantenibilidad y escalabilidad al permitir múltiples implementaciones concretas.

#### `comentario_dto.py`

- Contiene la clase ComentarioDTO, un objeto que encapsula los atributos de un comentario (id, texto, usuario\_email, calificación).
- Aplica el patrón **DTO (Data Transfer Object)**, el cual facilita el transporte de datos entre capas del sistema de forma segura, estructurada y desacoplada.
- Favorece la claridad y validación en la manipulación de datos de entrada y salida.

#### `conexion.py`

- Centraliza la lógica de conexión a los motores de base de datos configurados (MySQL o PostgreSQL).
- Utiliza los parámetros definidos en config.json, permitiendo establecer conexiones dinámicas según el entorno.
- Evita la duplicación de código al ofrecer un punto único para la gestión de conexiones.

#### `dao_factory.py`

- Define una clase abstracta DaoFactory con métodos generadores de DAOs, siguiendo el patrón **Abstract Factory**.
- Sirve como base para fábricas específicas, permitiendo instanciar componentes de acceso a datos sin depender de su implementación concreta.

#### `mysql_factory.py`

- Implementa DaoFactory para el caso del motor MySQL.
- Contiene la lógica necesaria para devolver instancias DAO compatibles con ese sistema de gestión de base de datos.

#### `postgres_factory.py`

- Similar a mysql\_factory.py, pero orientado a PostgreSQL.
- Permite que la capa de acceso a datos funcione correctamente en entornos con PostgreSQL, sin modificar el resto del sistema.

#### **get\_factory.py**

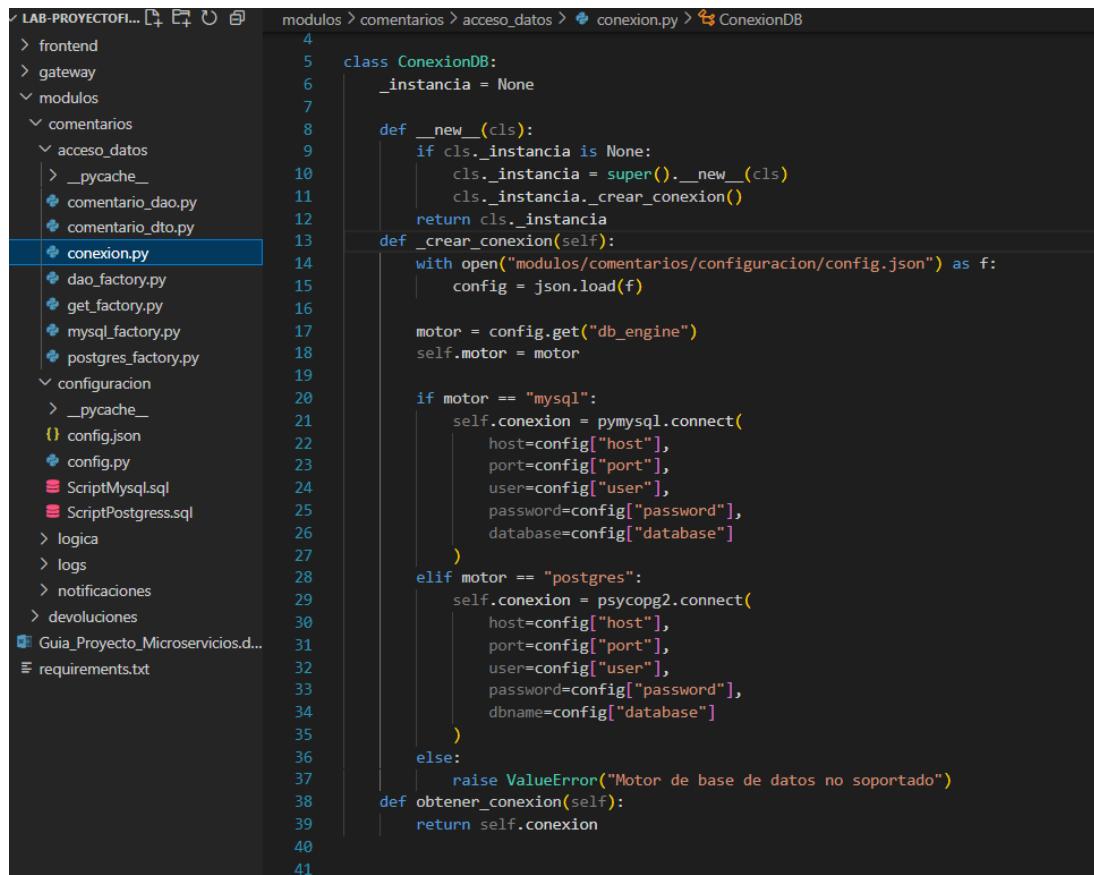
- Función utilitaria que consulta la configuración activa y retorna la fábrica correspondiente (MysqlFactory o PostgresFactory).
- Este componente es el punto de entrada para aplicar la inyección dinámica de dependencias según el motor de base de datos definido.

---

#### **✖ Principios arquitectónicos aplicados**

- **Separación de responsabilidades:** esta capa se enfoca exclusivamente en la interacción con la base de datos, manteniéndose independiente de la lógica de negocio y la presentación.
- **Desacoplamiento:** gracias al uso de patrones de diseño, los componentes superiores del sistema no dependen de clases concretas, sino de abstracciones.
- **Escalabilidad:** la arquitectura permite agregar nuevos motores de base de datos o nuevas entidades sin afectar la estructura existente.
- **Reusabilidad:** los DAOs, DTOs y fábricas pueden ser utilizados por múltiples módulos o microservicios.

## modulos\comentarios\acceso\_datos\conexión.py



```
LAB-PROYECTOFL... 4 modulos > comentarios > acceso_datos > connexion.py > ConexionDB
> frontend
> gateway
modulos
  < comentarios
    < acceso_datos
      > __pycache__
        comentario_dao.py
        comentario_dto.py
      connexion.py
        dao_factory.py
        get_factory.py
        mysql_factory.py
        postgres_factory.py
    < configuracion
      > __pycache__
        config.json
        config.py
        ScriptMysql.sql
        ScriptPostgress.sql
    > logica
    > logs
    > notificaciones
    > devoluciones
  Guia_Proyecto_Microservicios.d...
requirements.txt
  4 class ConexionDB:
  5     _instancia = None
  6
  7     def __new__(cls):
  8         if cls._instancia is None:
  9             cls._instancia = super().__new__(cls)
 10            cls._instancia._crear_conexion()
 11
 12     return cls._instancia
 13
 14     def _crearConexion(self):
 15         with open("modulos/comentarios/configuracion/config.json") as f:
 16             config = json.load(f)
 17
 18             motor = config.get("db_engine")
 19             self.motor = motor
 20
 21             if motor == "mysql":
 22                 self.conexion = pymysql.connect(
 23                     host=config["host"],
 24                     port=config["port"],
 25                     user=config["user"],
 26                     password=config["password"],
 27                     database=config["database"]
 28                 )
 29             elif motor == "postgres":
 30                 self.conexion = psycopg2.connect(
 31                     host=config["host"],
 32                     port=config["port"],
 33                     user=config["user"],
 34                     password=config["password"],
 35                     dbname=config["database"]
 36             )
 37             else:
 38                 raise ValueError("Motor de base de datos no soportado")
 39
 40             def obtenerConexion(self):
 41                 return self.conexion
```

Este archivo implementa la clase `ConexionDB`, encargada de crear y mantener una única instancia de conexión hacia la base de datos, utilizando un enfoque flexible y configurable para soportar distintos motores (MySQL y PostgreSQL). Su diseño aplica el patrón de diseño **Singleton**, junto con principios de configuración externa y desacoplamiento arquitectónico.

### Detalle del código y componentes clave

#### 1. Clase `ConexionDB` y atributo `_instancia`

- Define una variable de clase `_instancia = None`, la cual es utilizada para asegurar que **solo se cree una única instancia de conexión** durante toda la ejecución del sistema.
- Este es el núcleo del patrón **Singleton**, que garantiza un único punto de acceso compartido a un recurso crítico, como una conexión a base de datos.

#### 2. Método especial `__new__()`

- Esta sobrecarga del constructor permite controlar la creación de instancias.
- Si `cls._instancia` es `None`, se crea un nuevo objeto mediante `super().__new__(cls)` y se invoca internamente `_crearConexion()`.
- En futuras invocaciones, retorna la misma instancia ya creada, evitando múltiples conexiones simultáneas innecesarias o conflictivas.

3. **Método \_crear\_conexion()**
    - Abre y lee el archivo config.json para extraer los parámetros del motor de base de datos seleccionado.
    - Identifica el motor a través de config.get("db\_engine") y asigna sus credenciales (host, port, user, password, database).
    - Si el motor es "mysql", utiliza pymysql.connect(); si es "postgres", se emplea psycopg2.connect().
    - Si el motor especificado no es reconocido, lanza un ValueError, haciendo el sistema robusto frente a errores de configuración.
  4. **Método obtenerConexion()**
    - Expone una interfaz pública para acceder al objeto de conexión activo.
    - Otros componentes del sistema (por ejemplo, los DAOs) lo invocan para reutilizar la misma conexión persistente.
- 

#### **Patrón de diseño aplicado: Singleton**

- **Definición:** patrón de diseño creacional que restringe la instanciación de una clase a un solo objeto.
- **Aplicación en este contexto:**
  - Solo se inicializa una conexión a la base de datos por ejecución.
  - Aumenta la eficiencia evitando múltiples conexiones abiertas innecesariamente.
  - Mejora la consistencia del estado de la base de datos y facilita la depuración.
- **Ventajas:**
  - Centraliza el control de acceso a la base de datos.
  - Mejora el rendimiento en sistemas que operan con alto volumen de transacciones.

#### **Buenas prácticas arquitectónicas**

-  **Configuración externa:** los detalles de conexión no están embebidos en el código, sino definidos en config.json.
  -  **Polimorfismo condicional:** el sistema soporta múltiples motores de base de datos (MySQL, PostgreSQL) desde un mismo punto de acceso.
  -  **Mantenibilidad y escalabilidad:** es fácil extender el soporte a otros motores simplemente añadiendo nuevas condiciones dentro de \_crearConexion().
-

## modulos\comentarios\acceso\_datos\dao\_factory.py

The screenshot shows a file explorer on the left and a code editor on the right. The file explorer displays a project structure under 'LAB-PROYECTOIFI...'. The 'modulos' folder contains 'comentarios' and 'acceso\_datos'. Inside 'acceso\_datos', there are files: \_\_pycache\_\_, comentario\_dao.py, comentario\_dto.py, conexion.py, and dao\_factory.py (which is selected). Other files like get\_factory.py, mysql\_factory.py, and postgres\_factory.py are also listed. The code editor shows the content of dao\_factory.py:

```
1 class ComentarioDAOFactory:
2     def crear_dao(self):
3         raise NotImplementedError
```

Este archivo define una clase base ComentarioDAOFactory cuyo objetivo principal es estandarizar la creación de objetos DAO para la entidad Comentario. Su diseño sigue el patrón de diseño **Abstract Factory**, el cual facilita la creación de objetos relacionados sin especificar sus clases concretas.

### Clase ComentarioDAOFactory

- Representa una **fábrica abstracta** para producir objetos DAO que acceden a la base de datos.
- La clase no implementa ninguna lógica concreta, sino que define una interfaz que las subclases deben extender.

### Método crear\_dao(self)

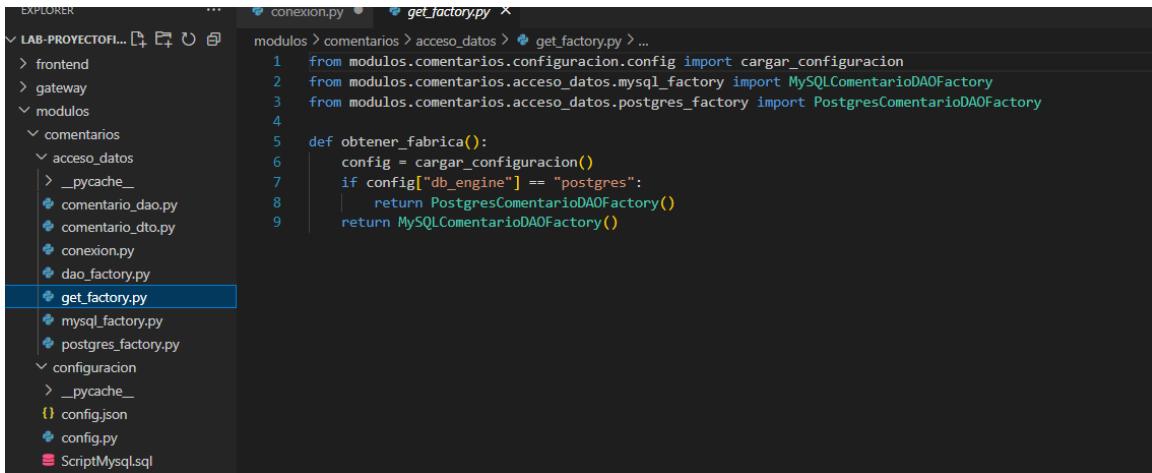
- Es un método que debe ser sobreescrito por cada implementación concreta.
- Al lanzar raise NotImplementedError, se fuerza a las subclases a proporcionar su propia implementación de este método.
- Sirve como contrato que garantiza consistencia entre las distintas fábricas concretas (por ejemplo, MySQL y PostgreSQL).

### ✿ Patrón de diseño aplicado: Abstract Factory

- **Definición:** patrón creacional que permite producir familias de objetos relacionados sin depender de sus clases concretas.
- **Aplicación en este archivo:**
  - ComentarioDAOFactory define la interfaz para crear objetos DAO.
  - Subclases como MysqlFactory y PostgresFactory implementan este método para retornar instancias de DAOs concretos (ComentarioDaoMysql, ComentarioDaoPostgres, etc.).
- **Ventajas:**
  - Promueve el **principio de inversión de dependencias**, ya que las capas superiores (como la lógica del microservicio) dependen de abstracciones y no de implementaciones concretas.

- Facilita la **extensibilidad** del sistema: agregar un nuevo tipo de DAO (por ejemplo, para Oracle o SQLite) solo requiere crear una nueva subclase de fábrica.
- Permite cambiar el motor de base de datos sin modificar el resto del sistema, lo que favorece el mantenimiento y la adaptabilidad.

## modulos\comentarios\acceso\_datos\get\_factory.py



```

1  from modulos.comentarios.configuracion.config import cargar_configuracion
2  from modulos.comentarios.acceso_datos.mysql_factory import MySQLComentarioDAOFactory
3  from modulos.comentarios.acceso_datos.postgres_factory import PostgresComentarioDAOFactory
4
5  def obtener_fabrica():
6      config = cargar_configuracion()
7      if config["db_engine"] == "postgres":
8          return PostgresComentarioDAOFactory()
9      return MySQLComentarioDAOFactory()

```

El archivo `get_factory.py` cumple la función de **resolver dinámicamente qué implementación concreta de DAO Factory debe usarse**, basándose en el motor de base de datos configurado en `config.json`. Actúa como un componente auxiliar dentro del patrón **Abstract Factory**, pero no define una clase propia porque su responsabilidad es puntual, estática y no requiere mantenimiento de estado.

### Importaciones clave

- `cargar_configuracion`: función definida en `config.py` que carga el archivo `config.json`.
- `MySQLComentarioDAOFactory` y `PostgresComentarioDAOFactory`: clases concretas de fábricas DAO especializadas según el motor de base de datos (MySQL o PostgreSQL).

### Función `obtener_fabrica()`

- Esta función es el núcleo del archivo. Lee el archivo de configuración para determinar qué motor de base de datos está en uso ("db\_engine").
- Si el motor es "postgres", retorna una instancia de `PostgresComentarioDAOFactory`. En cualquier otro caso (por defecto), retorna `MySQLComentarioDAOFactory`.

### No se define una clase porque:

- **No se necesita mantener estado ni atributos**: la lógica de decisión es completamente funcional (sin contexto ni almacenamiento interno).
- **Simplicidad y legibilidad**: el patrón de uso previsto es importar directamente `obtener_fabrica()` y llamarlo para recibir una fábrica DAO.
- **Responsabilidad única**: su única responsabilidad es entregar la fábrica apropiada; no administra recursos ni define jerarquías.

### Integración con el patrón Abstract Factory

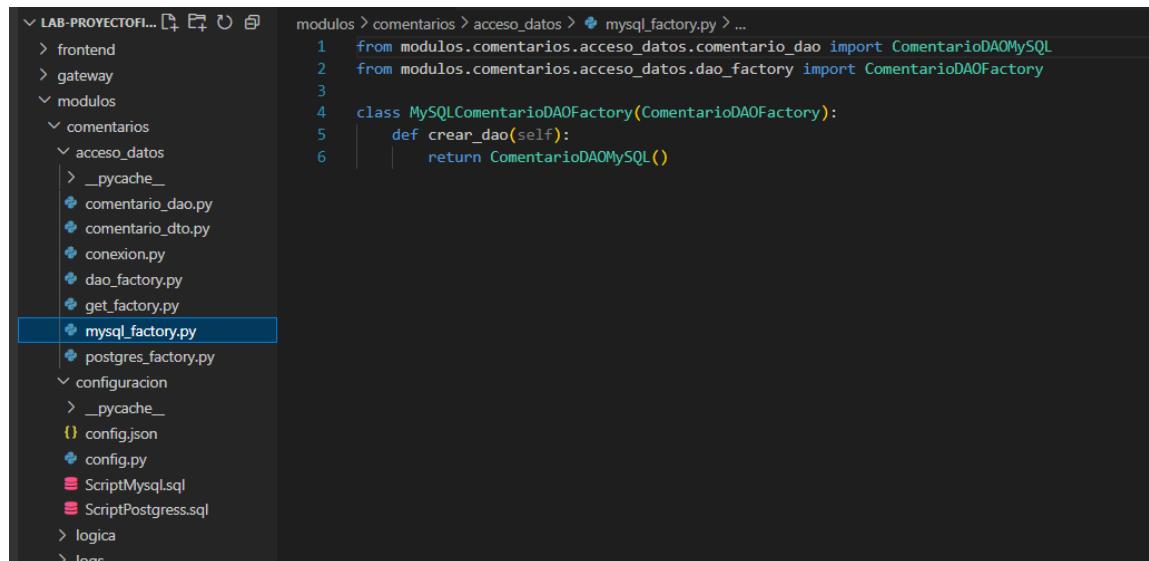
- Este archivo **no implementa el patrón en sí**, sino que actúa como **componente de orquestación** que selecciona cuál implementación concreta del patrón Abstract Factory será utilizada en tiempo de ejecución.

- Es decir, permite aplicar el **principio de inversión de dependencias**, delegando la decisión del tipo concreto de fábrica al entorno de configuración, y no al código de negocio.

#### Ventajas del diseño funcional sin clase

- **Facilita la inyección de dependencias** sin necesidad de instanciar objetos auxiliares.
- **Menor complejidad**: al no usar clases innecesarias, se evita sobreestructurar una función que cumple un propósito único.
- **Claridad y cohesión**: el diseño comunica claramente su intención: elegir y retornar la fábrica adecuada en función del entorno.

## modulos\comentarios\acceso\_datos\mysql\_factory.py



The screenshot shows a file explorer on the left and a code editor on the right. The file explorer displays a project structure:

```

LAB-PROYECTOFI... └── modulos
    ├── frontend
    ├── gateway
    └── comentarios
        ├── acceso_datos
        │   ├── __pycache__
        │   ├── comentario_dao.py
        │   ├── comentario_dto.py
        │   ├── conexion.py
        │   ├── dao_factory.py
        │   ├── get_factory.py
        │   ├── mysql_factory.py
        │   └── postgres_factory.py
        └── configuracion
            ├── __pycache__
            ├── config.json
            ├── config.py
            ├── ScriptMySQL.sql
            └── ScriptPostgres.sql
    └── logica
    └── logs

```

The code editor shows the content of `mysql_factory.py`:

```

modulos > comentarios > acceso_datos > mysql_factory.py ...
1  from modulos.comentarios.acceso_datos.comentario_dao import ComentarioDAOMySQL
2  from modulos.comentarios.acceso_datos.dao_factory import ComentarioDAOFactory
3
4  class MySQLComentarioDAOFactory(ComentarioDAOFactory):
5      def crear_dao(self):
6          return ComentarioDAOMySQL()

```

Este archivo implementa una **fábrica concreta** dentro del patrón **Abstract Factory**, responsable de generar instancias DAO específicas para el motor de base de datos MySQL. Su función es separar la lógica de creación de objetos DAO de la lógica de negocio, favoreciendo así el desacoplamiento y la extensibilidad del sistema.

#### Desglose del código

##### 1. Importaciones

- ComentarioDAOMySQL: clase DAO que implementa las operaciones CRUD usando MySQL.
- ComentarioDAOFactory: clase base abstracta que define el contrato del método `crear_dao()`.

##### 2. Clase MySQLComentarioDAOFactory

- Es una subclase concreta de ComentarioDAOFactory.

- Especializada en retornar un DAO configurado para el motor de base de datos MySQL.
3. **Método crear\_dao()**
- Sobrescribe el método abstracto de la clase base.
  - Devuelve una nueva instancia de ComentarioDAOMySQL, encapsulando así la lógica de creación del DAO correspondiente al motor seleccionado.
  - Este método es invocado desde el sistema cuando se desea obtener un objeto DAO adaptado a MySQL, sin necesidad de conocer detalles de implementación.

### Patrón de diseño aplicado: Abstract Factory

- Este archivo representa la implementación concreta de una **fábrica** dentro del patrón **Abstract Factory**.
- Permite crear un objeto DAO específico para MySQL sin acoplar el código consumidor al tipo concreto.
- Facilita cambiar de base de datos (por ejemplo, a PostgreSQL) simplemente cambiando la fábrica que se utiliza, sin modificar el resto del sistema.

### Ventajas del diseño aplicado

- **Desacoplamiento**: el consumidor del DAO no depende de ComentarioDAOMySQL, sino de una fábrica que cumple un contrato.
- **Escalabilidad**: se pueden añadir nuevas fábricas (Oracle, SQLite, etc.) sin modificar este archivo.
- **Simplicidad de configuración**: combinando con get\_factory.py, se selecciona dinámicamente qué fábrica usar según el archivo config.json.

## modulos\comentarios\acceso\_datos\ComentarioDTO.py



```

1  from datetime import datetime
2
3  class ComentarioDTO:
4      def __init__(self, id=None, texto="", usuario_email="", calificacion=0, fecha=None):
5          self.id = id
6          self.texto = texto
7          self.usuario_email = usuario_email
8          self.calificacion = calificacion
9          self.fecha = fecha or datetime.now()
10
11     def __str__(self):
12         return f"ComentarioDTO(id={self.id}, texto='{self.texto}', usuario_email='{self.usuario_email}', calificacion={self.calificacion}, fecha={self.fecha})"
13
14

```

Este archivo define la clase ComentarioDTO, que encapsula los atributos esenciales de un comentario y los transporta entre capas del sistema (por ejemplo, desde la API hasta la base de datos o desde el DAO hacia la interfaz). Esta clase implementa el patrón de diseño **DTO (Data Transfer Object)**, ampliamente utilizado en arquitecturas multicapa y de microservicios para facilitar el transporte seguro y estructurado de datos.

## Detalle del código

1. Importación de `datetime`
  - Se importa `datetime.now()` para generar una marca de tiempo si no se especifica una fecha explícita al crear un comentario.
2. Clase `ComentarioDTO`
  - Representa un objeto de datos puro, sin lógica de negocio.
  - Su único propósito es **transportar datos** de manera coherente entre componentes.
3. Método `__init__()`
  - Recibe como parámetros:
    - id: identificador único (puede ser `None` si es un nuevo comentario).
    - texto: el contenido del comentario.
    - usuario\_email: correo del usuario que comenta.
    - calificación: valoración asociada (entero).
    - fecha: opcional; si no se proporciona, se asigna la fecha y hora actuales.
  - Asigna los valores a atributos de instancia, creando una estructura clara y reusable para representar un comentario en tránsito.
4. Método `__str__()`
  - Define una representación legible de la instancia, útil para depuración o logging.
  - Retorna una cadena con todos los atributos del objeto.

---

## Patrón aplicado: DTO (Data Transfer Object)

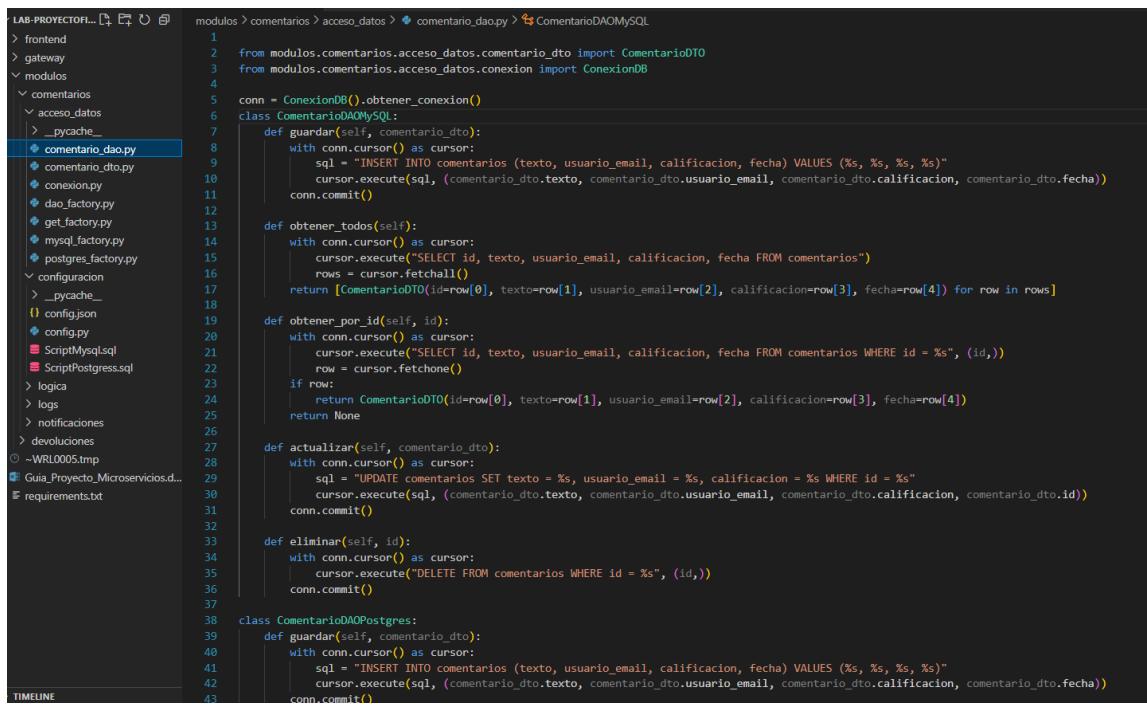
- **Objetivo:** transportar datos entre distintas capas o servicios de forma segura y eficiente.
- **Ventajas:**
  - Evita el acoplamiento directo a entidades de base de datos o modelos internos.
  - Centraliza y controla los atributos permitidos en una operación.
  - Mejora la claridad del sistema y la trazabilidad de los datos en tránsito.
  - Facilita pruebas, validación y serialización (por ejemplo, a JSON).

---

## Buenas prácticas

- **Inicialización con valores por defecto**, lo que permite crear objetos flexibles y seguros ante entradas incompletas.
  - **Representación explícita del objeto (`__str__`)**, que mejora la legibilidad durante pruebas o trazas.
  - **Compatibilidad con frameworks**: este DTO puede serializarse fácilmente en APIs REST usando FastAPI, Flask, etc.
-

## modulos\comentarios\acceso\_datos\comentario\_dao.py



```
LAB-PROYECTOFL... □ □ □ □
> frontend
> gateway
└ modulos
  └ comentarios
    └ acceso_datos
      > _pycache_
        < comentario_dao.py >
        < comentario_dto.py >
        < conexion.py >
        < dao_factory.py >
        < get_factory.py >
        < mysql_factory.py >
        < postgres_factory.py >
      < configuracion >
        > _pycache_
        < config.json >
        < config.py >
        < ScriptMySQL.sql >
        < ScriptPostgres.sql >
      < logica >
      < logs >
      < notificaciones >
      < devoluciones >
      ~WRL005.tmp
      < Guia_Proyecto_Microservicios.d...
      requirements.txt
TIMELINE
```

```
1  from modulos.comentarios.acceso_datos.comentario_dto import ComentarioDTO
2  from modulos.comentarios.acceso_datos.conexion import ConexionDB
3
4
5  conn = ConexionDB().obtenerConexion()
6  class ComentarioDAOMySQL:
7
8      def guardar(self, comentario_dto):
9          with conn.cursor() as cursor:
10             sql = "INSERT INTO comentarios (texto, usuario_email, calificacion, fecha) VALUES (%s, %s, %s, %s)"
11             cursor.execute(sql, (comentario_dto.texto, comentario_dto.usuario_email, comentario_dto.calificacion, comentario_dto.fecha))
12             conn.commit()
13
14      def obtener_todos(self):
15          with conn.cursor() as cursor:
16              cursor.execute("SELECT id, texto, usuario_email, calificacion, fecha FROM comentarios")
17              rows = cursor.fetchall()
18              return [ComentarioDTO(id=row[0], texto=row[1], usuario_email=row[2], calificacion=row[3], fecha=row[4]) for row in rows]
19
20      def obtener_por_id(self, id):
21          with conn.cursor() as cursor:
22              cursor.execute("SELECT id, texto, usuario_email, calificacion, fecha FROM comentarios WHERE id = %s", (id,))
23              row = cursor.fetchone()
24              if row:
25                  return ComentarioDTO(id=row[0], texto=row[1], usuario_email=row[2], calificacion=row[3], fecha=row[4])
26              return None
27
28      def actualizar(self, comentario_dto):
29          with conn.cursor() as cursor:
30              sql = "UPDATE comentarios SET texto = %s, usuario_email = %s, calificacion = %s WHERE id = %s"
31              cursor.execute(sql, (comentario_dto.texto, comentario_dto.usuario_email, comentario_dto.calificacion, comentario_dto.id))
32              conn.commit()
33
34      def eliminar(self, id):
35          with conn.cursor() as cursor:
36              cursor.execute("DELETE FROM comentarios WHERE id = %s", (id,))
37              conn.commit()
38
39      class ComentarioDAOPostgres:
40          def guardar(self, comentario_dto):
41              with conn.cursor() as cursor:
42                  sql = "INSERT INTO comentarios (texto, usuario_email, calificacion, fecha) VALUES (%s, %s, %s, %s)"
43                  cursor.execute(sql, (comentario_dto.texto, comentario_dto.usuario_email, comentario_dto.calificacion, comentario_dto.fecha))
44                  conn.commit()
```

Este archivo contiene las clases ComentarioDAOMySQL y ComentarioDAOPostgres, que implementan las operaciones de acceso a la base de datos para el módulo de comentarios. Ambas clases siguen el patrón **DAO (Data Access Object)**, el cual encapsula todas las operaciones CRUD y oculta los detalles de interacción con el sistema de almacenamiento.

### ✿ Detalle del código

#### 1. Importaciones

- ComentarioDTO: clase de transporte de datos que representa un comentario.
- ConexionDB: clase Singleton que proporciona una conexión reutilizable a la base de datos activa.

#### 2. Conexión compartida

- La conexión se obtiene una sola vez mediante ConexionDB().obtenerConexion(), lo que favorece el control de recursos y eficiencia en la comunicación con la base de datos.

#### 3. Clase ComentarioDAOMySQL

- Contiene los métodos:
  - guardar(): inserta un nuevo comentario.
  - obtener\_todos(): consulta todos los comentarios.
  - obtener\_por\_id(): consulta un comentario por su ID.
  - actualizar(): modifica un comentario existente.
  - eliminar(): elimina un comentario por ID.
- Todos los métodos utilizan SQL específico para MySQL, aunque en este caso es compatible también con PostgreSQL (por el uso de placeholders %s).

#### 4. Clase ComentarioDAOPostgres

- Implementa exactamente los mismos métodos que ComentarioDAOMySQL.
  - Utiliza las mismas estructuras SQL, lo cual refleja un diseño **polimórfico**: distintas clases, mismas operaciones, comportamiento adaptado según el contexto de ejecución.
- 

#### Patrón aplicado: DAO (Data Access Object)

- El patrón DAO permite separar la lógica de negocio de la lógica de persistencia.
  - Encapsula completamente las operaciones de base de datos, facilitando el mantenimiento y la prueba de cada componente de forma aislada.
- 

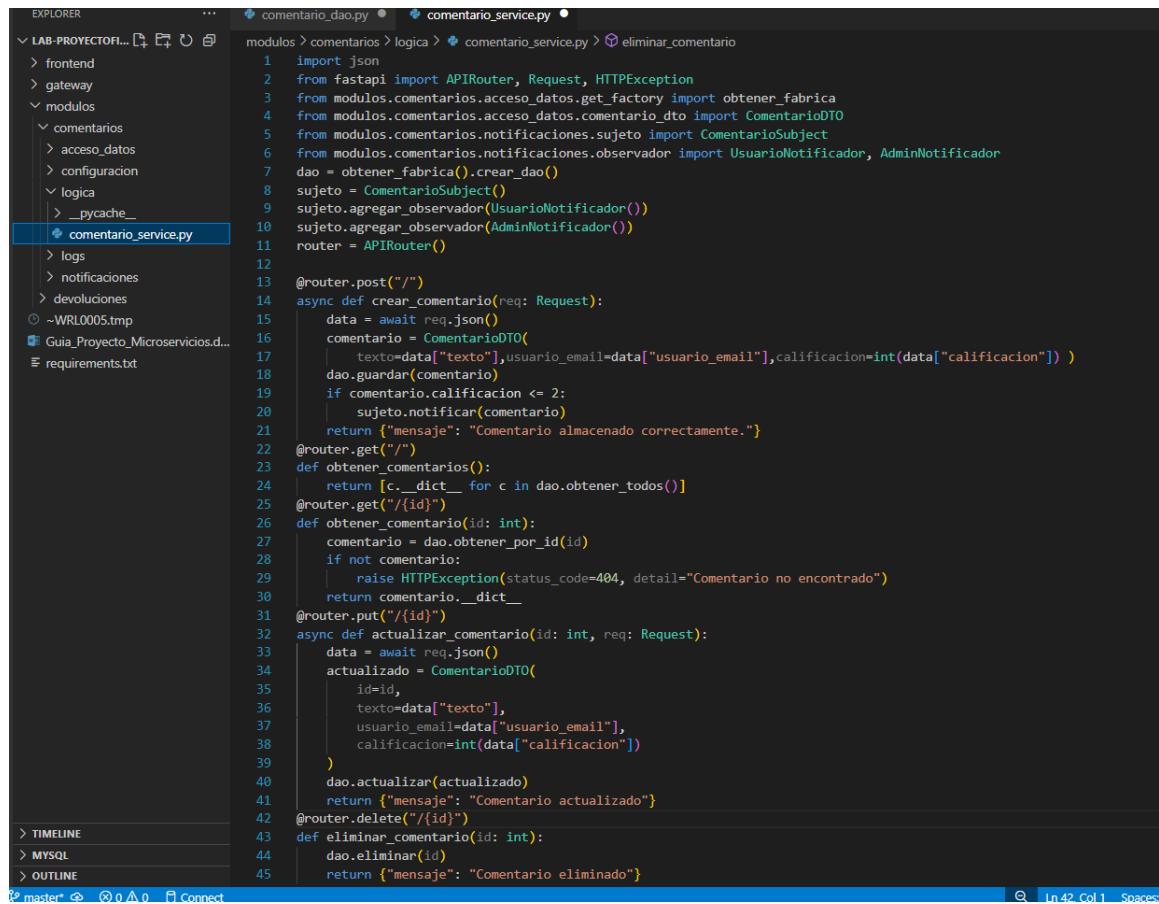
#### Reutilización y consistencia entre motores

- Las clases ComentarioDAOMySQL y ComentarioDAOPostgres implementan la **misma interfaz funcional**.
  - Esto permite a la lógica del sistema (por ejemplo, en comentario\_service.py) utilizar cualquier implementación DAO sin necesidad de modificar su código.
  - La decisión del motor se toma en tiempo de ejecución mediante el patrón **Abstract Factory** (ver mysql\_factory.py, postgres\_factory.py).
- 

#### Ventajas

- **Reutilización del código**: al mantener la misma estructura de métodos, se minimiza la duplicación.
- **Escalabilidad tecnológica**: el sistema puede migrar de motor de base de datos o soportar múltiples motores sin modificar la lógica de alto nivel.
- **Claridad modular**: cada clase DAO está contenida y es fácilmente identifiable por su propósito y motor.

## modulos\comentarios\logica\comentario\_service.py



```
1 import json
2 from fastapi import APIRouter, Request, HTTPException
3 from modulos.comentarios.acceso_datos.get_factory import obtener_fabrica
4 from modulos.comentarios.acceso_datos.comentario_dto import ComentarioDTO
5 from modulos.comentarios.notificaciones.sujeto import ComentarioSubject
6 from modulos.comentarios.notificaciones.observador import UsuarioNotificador, AdminNotificador
7 dao = obtener_fabrica().crear_dao()
8 sujeto = ComentarioSubject()
9 sujeto.agregar_observador(UsuarioNotificador())
10 sujeto.agregar_observador(AdminNotificador())
11 router = APIRouter()
12
13 @router.post("/")
14 async def crear_comentario(req: Request):
15     data = await req.json()
16     comentario = ComentarioDTO(
17         texto=data["texto"], usuario_email=data["usuario_email"], calificacion=int(data["calificacion"]))
18     dao.guardar(comentario)
19     if comentario.calificacion <= 2:
20         sujeto.notificar(comentario)
21     return {"mensaje": "Comentario almacenado correctamente."}
22 @router.get("/")
23 def obtener_comentarios():
24     return [c._dict_ for c in dao.obtener.todos()]
25 @router.get("/{id}")
26 def obtener_comentario(id: int):
27     comentario = dao.obtener_por_id(id)
28     if not comentario:
29         raise HTTPException(status_code=404, detail="Comentario no encontrado")
30     return comentario._dict_
31 @router.put("/{id}")
32 async def actualizar_comentario(id: int, req: Request):
33     data = await req.json()
34     actualizado = ComentarioDTO(
35         id=id,
36         texto=data["texto"],
37         usuario_email=data["usuario_email"],
38         calificacion=int(data["calificacion"]))
39     dao.actualizar(actualizado)
40     return {"mensaje": "Comentario actualizado"}
41 @router.delete("/{id}")
42 def eliminar_comentario(id: int):
43     dao.eliminar(id)
44     return {"mensaje": "Comentario eliminado"}
```

### Objetivo

- Expone la API del microservicio de comentarios mediante FastAPI.
- Orquesta el flujo de entrada y salida entre los clientes, la capa de datos y el sistema de notificaciones.

### Componentes clave

- dao = obtener\_fabrica().crear\_dao(): inyección dinámica del DAO según configuración.
- ComentarioSubject: instancia que implementa el patrón Observer para notificar observadores ante eventos (comentarios negativos).
- router = APIRouter(): define las rutas REST específicas del microservicio.

### Rutas implementadas

- POST /: crea un comentario. Si la calificación ≤ 2, activa notificación vía observadores.
- GET /: retorna todos los comentarios.
- GET /{id}: retorna un comentario por su ID. Lanza error 404 si no existe.
- PUT /{id}: actualiza un comentario existente.
- DELETE /{id}: elimina un comentario por ID.

### Patrones aplicados

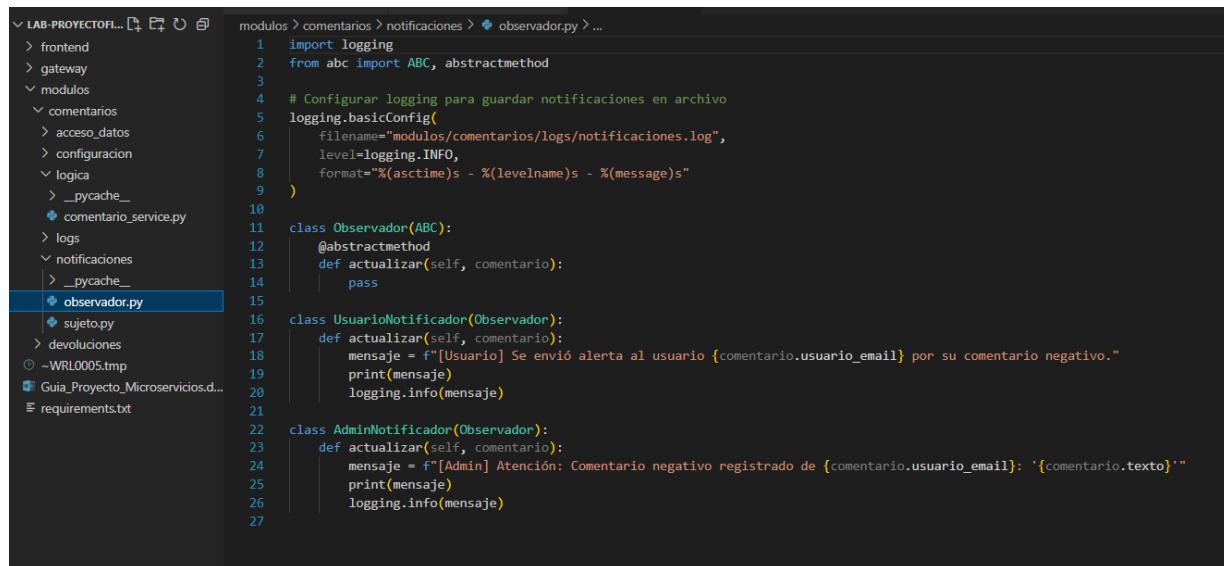
- **Inyección de dependencias:** el DAO se selecciona mediante una fábrica.

- **Observer:** se notifican subsistemas cuando se detectan comentarios críticos.
- **DTO:** ComentarioDTO se usa para trasladar datos de manera estructurada entre capas.

### **Ventajas**

- Código desacoplado, reutilizable y extensible.
- Compatible con múltiples motores de base de datos.
- Preparado para monitoreo y escalabilidad en entorno microservicio.

## modulos\comentarios\noticaciones\observador.py



The screenshot shows a code editor with a file tree on the left and the content of `observador.py` on the right.

**File Tree:**

- LAB-PROYECTO...
- frontend
- gateway
- modulos
  - comentarios
    - acceso\_datos
    - configuracion
    - logica
      - \_pycache\_
      - comentario\_service.py
    - logs
    - notificaciones
      - \_pycache\_
      - observador.py
      - sujeto.py
  - devoluciones
  - ~WRL0005.tmp
  - Guia\_Proyecto\_Microservicios.d...
  - requirements.txt

### 1. Propósito

- Este archivo define el mecanismo de notificación ante eventos críticos (comentarios negativos).
- Implementa el patrón **Observer**, que permite notificar múltiples objetos interesados sin acoplarlos directamente al productor de eventos.

### 2. Componentes principales

- Observador: clase abstracta con el método `actualizar()`, que define el contrato para todos los observadores.
- UsuarioNotificador: observador que emite una alerta dirigida al usuario cuando se detecta una calificación negativa.
- AdminNotificador: observador que genera una notificación dirigida al administrador con el detalle del comentario crítico.

### 3. Mecanismo de notificación

- Ambos observadores implementan `actualizar()`, construyendo un mensaje y registrándolo:
  - `print()`: salida inmediata en consola.
  - `logging.info()`: guarda el mensaje en un archivo de log ubicado en `logs/notificaciones.log`.

### 4. Configuración de logging

- Se configura el sistema de logging para registrar notificaciones con formato estructurado y nivel INFO.

### 5. Ventajas del patrón aplicado

- Desacoplamiento**: el servicio de comentarios no conoce a detalle quién recibe la notificación.
- Extensibilidad**: se pueden agregar nuevos observadores (ej. SMSNotificador, SlackNotificador) sin modificar la lógica existente.
- Reusabilidad**: los observadores pueden ser reutilizados por otros microservicios si se requiere lógica de reacción similar.

## modulos\comentarios\noticiones\sujeto.py

The screenshot shows a code editor interface with a sidebar and a main code view. The sidebar on the left displays a project tree under 'LAB-PROYECTOFI...'. The main area shows the content of 'sujeto.py'.

```
1  class ComentarioSubject:
2      def __init__(self):
3          self.observadores = []
4
5      def agregar_observador(self, observador):
6          self.observadores.append(observador)
7
8      def notificar(self, comentario):
9          for observador in self.observadores:
10             observador.actualizar(comentario)
```

### 1. Propósito

- Define la clase ComentarioSubject, que representa el **sujeto** observado en el patrón Observer.
- Administra la lista de observadores y se encarga de notificarles cuando ocurre un evento relevante (por ejemplo, un comentario negativo).

### 2. Componentes clave

- self.observadores: lista que almacena los objetos que implementan la interfaz Observador.
- agregar\_observador(): permite registrar dinámicamente nuevos observadores.
- notificar(): recorre la lista de observadores y llama su método actualizar() con el comentario como parámetro.

### 3. Integración con el sistema

- Es utilizado por comentario\_service.py para emitir notificaciones cuando se detectan comentarios con calificación ≤ 2.
- Se integra con clases como UsuarioNotificador y AdminNotificador, que reaccionan ante los eventos propagados.

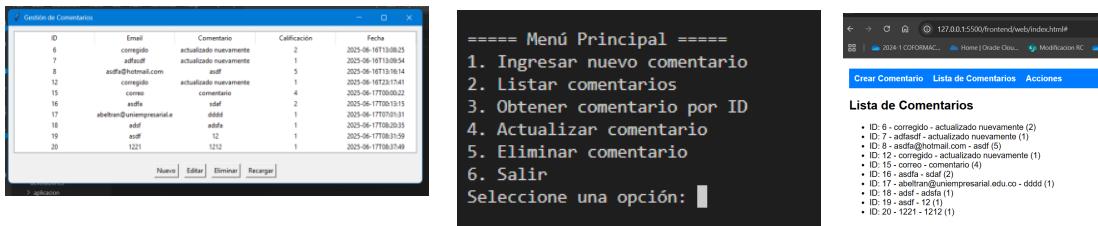
#### ✿ Patrón aplicado: Observer (parte del "Subject")

- ComentarioSubject es el emisor de eventos (subject).
- Invoca el método actualizar() en cada observador registrado cuando se llama a notificar().

#### ✓ Ventajas del diseño

- **Extensible**: se pueden agregar observadores sin modificar el sujeto.
- **Bajo acoplamiento**: el sujeto no necesita conocer los detalles internos de cada observador.
- **Flexible**: se pueden activar múltiples respuestas ante un mismo evento de manera paralela.

## Frontend



► Interfaz Web

The browser address bar shows the URL 127.0.0.1:5500/frontend/web/index.html#. The page has a blue header bar with three tabs: 'Crear Comentario', 'Lista de Comentarios', and 'Acciones'. Below the header, the text 'Lista de Comentarios' is displayed. A list of comments is shown, which matches the data from the first screenshot.

## Listado de Comentarios

- ID: 6 - corregido - actualizado nuevamente (2)
- ID: 7 - adfasdf - actualizado nuevamente (1)
- ID: 8 - asdfa@hotmail.com - asdf (5)
- ID: 12 - corregido - actualizado nuevamente (1)
- ID: 15 - correo - comentario (4)
- ID: 16 - asdfa - sdfa (2)
- ID: 17 - abeltran@uniempresarial.edu.co - dddd (1)
- ID: 18 - adsf - adsfa (1)
- ID: 19 - asdf - 12 (1)
- ID: 20 - 1221 - 1212 (1)

## frontend\web\index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <title>Gestión de Comentarios</title>
    <link rel="stylesheet" href="estilos.css" />
</head>
<body>
    <div class="navbar">
        <a href="#" onclick="mostrarSeccion('crear')">Crear Comentario</a>
```

```

        <a href="#" onclick="mostrarSeccion('lista')">Lista de Comentarios</a>
        <a href="#" onclick="mostrarSeccion('acciones')">Acciones</a>
    </div>

    <div id="crear" class="seccion">
        <h2>Crear Comentario</h2>
        <form id="formComentario">
            <textarea id="texto" placeholder="Comentario"></textarea>
            <input type="text" id="email" placeholder="Correo del usuario" />
            <input type="number" id="calificacion" placeholder="Calificación (1-5)" />
            <button type="submit">Guardar</button>
        </form>
    </div>

    <div id="lista" class="seccion">
        <h2>Lista de Comentarios</h2>
        <ul id="listaComentarios"></ul>
    </div>

    <div id="acciones" class="seccion">
        <h2>Buscar / Actualizar / Eliminar</h2>
        <input type="text" id="idBuscar" placeholder="ID del comentario" />
        <button onclick="buscarComentario()">Buscar</button>
        <textarea id="textoAccion" placeholder="Comentario"></textarea>
        <input type="text" id="emailAccion" placeholder="Correo del usuario" />
        <input type="number" id="calificacionAccion" placeholder="Calificación (1-5)" />
        <button onclick="actualizarComentario()">Actualizar</button>
        <button onclick="eliminarComentario()">Eliminar</button>
    </div>

    <script src="config.js"></script>
    <script src="app.js"></script>
</body>
</html>

```

## Estructura general del documento

- Se declara el tipo de documento con `<!DOCTYPE html>`, asegurando compatibilidad con navegadores modernos.
- El atributo `lang="es"` especifica el idioma del contenido como español, lo cual favorece la accesibilidad y posicionamiento en buscadores.

## Encabezado (`<head>`)

- `<meta charset="UTF-8" />`: garantiza el soporte de caracteres especiales como tildes y símbolos propios del idioma español.
- `<title>` define el nombre que aparecerá en la pestaña del navegador: *Gestión de Comentarios*.
- `<link rel="stylesheet" href="estilos.css" />`: vincula una hoja de estilos externa que define el diseño visual del sitio, fomentando separación entre estructura y presentación.

## Barra de navegación (`<div class="navbar">`)

- Contiene enlaces (`<a>`) que ejecutan funciones JavaScript mediante el atributo `onclick`.
- Estas funciones (`mostrarSeccion('crear')`, etc.) alternan la visibilidad de las secciones sin necesidad de recargar la página.
- Aplica el principio de navegación dinámica de una SPA (Single Page Application).

## Sección: Crear Comentario

- `<form id="formComentario">` contiene campos de entrada para:
  - El texto del comentario (`<textarea id="texto">`),

- El correo del usuario (<input type="text" id="email">),
  - La calificación (<input type="number" id="calificacion">).
- El botón "Guardar" enviará los datos al backend mediante JavaScript utilizando fetch().

#### Sección: Lista de Comentarios

- <ul id="listaComentarios"> está vacía inicialmente.
- Será poblada dinámicamente desde JavaScript al consultar el microservicio.
- Cada comentario se mostrará como un elemento <li> con sus atributos principales.

#### Sección: Buscar / Actualizar / Eliminar Comentarios

- Incluye campos adicionales para realizar operaciones de mantenimiento sobre comentarios:
  - Buscar por ID (<input id="idBuscar">).
  - Modificar texto, email y calificación (textarea, input).
- Botones asociados ejecutan funciones: buscarComentario(), actualizarComentario(), eliminarComentario().

#### Carga de scripts

- <script src="config.js"> contiene la configuración base, como las URLs del backend.
- <script src="app.js"> implementa la lógica de interacción: captura de eventos, llamadas fetch(), manipulación del DOM.

#### Principios aplicados

- **Separación de responsabilidades:** HTML define la estructura, CSS el diseño y JS la lógica.
- **Interactividad dinámica:** sin recargar la página, gracias al uso de JavaScript.
- **Modularidad y mantenibilidad:** permite escalar la aplicación fácilmente.
- **Integración con microservicios:** actúa como cliente REST para consumir servicios ofrecidos por el backend

## frontend\web\estilos.css

```
.navbar {
    background-color: #007bff;
    padding: 10px;
}
.navbar a {
    color: white;
    margin-right: 15px;
    text-decoration: none;
}
.seccion {
    display: none;
}
```

- El CSS aplica un diseño simple y funcional que mejora la experiencia del usuario.
  - Se usa una barra de navegación (.navbar) con enlaces visibles y una paleta de colores moderna.
  - Cada sección se oculta por defecto (display: none), y se muestra dinámicamente según la acción del usuario.
- 

## frontend\web\app.js

```

async function listarComentarios() {
    const res = await fetch(API_BASE + ENDPOINTS.read_all);
    const data = await res.json();
    const lista = document.getElementById("listaComentarios");
    lista.innerHTML = "";
    data.forEach(c => {
        const item = document.createElement("li");
        item.textContent = `ID: ${c.id} - ${c.usuario_email} - ${c.texto} (${c.calificacion})`;
        lista.appendChild(item);
    });
}

document.getElementById("formComentario").addEventListener("submit", async function(e) {
    e.preventDefault();
    const body = {
        texto: document.getElementById("texto").value,
        usuario_email: document.getElementById("email").value,
        calificacion: parseInt(document.getElementById("calificacion").value)
    };
    await fetch(API_BASE + ENDPOINTS.create, {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify(body)
    });
    alert("Comentario creado.");
    listarComentarios();
    mostrarSeccion('lista');
});

async function buscarComentario() {
    const id = document.getElementById("idBuscar").value;
    const res = await fetch(API_BASE + ENDPOINTS.read_one.replace("{id}", id));
    if (res.ok) {
        const data = await res.json();
        document.getElementById("textoAccion").value = data.texto;
        document.getElementById("emailAccion").value = data.usuario_email;
        document.getElementById("calificacionAccion").value = data.calificacion;
        mostrarSeccion('acciones');
        alert("Comentario cargado para edición.");
    } else {
        alert("Comentario no encontrado.");
    }
}

async function actualizarComentario() {
    const id = document.getElementById("idBuscar").value;
    const body = {
        texto: document.getElementById("textoAccion").value,
        usuario_email: document.getElementById("emailAccion").value,
        calificacion: parseInt(document.getElementById("calificacionAccion").value)
    };
}

```

```

};

const res = await fetch(API_BASE + ENDPOINTS.update.replace("{id}", id), {
  method: "PUT",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify(body)
});
const result = await res.json();
alert(result.mensaje || "Actualizado");
listarComentarios();
mostrarSeccion('lista');
}

async function eliminarComentario() {
  const id = document.getElementById("idBuscar").value;
  const res = await fetch(API_BASE + ENDPOINTS.delete.replace("{id}", id), { method: "DELETE"
});
  const result = await res.json();
  alert(result.mensaje || "Eliminado");
  listarComentarios();
  mostrarSeccion('lista');
}

function mostrarSeccion(id) {
  document.querySelectorAll(".seccion").forEach(s => s.style.display = "none");
  document.getElementById(id).style.display = "block";
}

listarComentarios();
mostrarSeccion('crear');

```

### 1. **listarComentarios()**

- Es una función asíncrona que consulta todos los comentarios disponibles.
- Utiliza fetch() para realizar una solicitud GET al endpoint definido en API\_BASE + ENDPOINTS.read\_all.
- Transforma la respuesta a formato JSON con await res.json().
- Limpia el contenido actual del <ul id="listaComentarios"> y lo repuebla dinámicamente.
- Por cada comentario recibido, crea un elemento <li> con el ID, correo, texto y calificación, y lo añade a la lista.

### 2. **Manejo del formulario de creación**

- El evento submit del formulario formComentario es interceptado con e.preventDefault() para evitar el comportamiento por defecto.
- Se extraen los valores ingresados por el usuario: texto del comentario, email y calificación.
- Se construye un objeto body con esos datos, el cual se envía al servidor como JSON mediante una solicitud POST.
- Al completarse la operación, se muestra un mensaje emergente (alert) y se actualiza la lista de comentarios invocando listarComentarios().
- Luego, se redirige visualmente a la sección de lista con mostrarSeccion('lista').

### 3. **buscarComentario()**

- Recupera el ID digitado en el campo idBuscar.
- Realiza una solicitud GET al backend usando el endpoint para obtener un comentario específico (read\_one).
- Si la respuesta es válida (res.ok), carga los datos del comentario en los campos correspondientes de la sección de acciones.
- También cambia la vista activa a dicha sección e informa al usuario que el comentario fue cargado.

- Si no se encuentra, muestra una alerta de error.
  - 4. **actualizarComentario()**
    - Obtiene el ID y los nuevos valores desde la interfaz (texto, correo, calificación).
    - Realiza una solicitud PUT al servidor, reemplazando {id} por el ID específico en el endpoint.
    - Envía los datos modificados en formato JSON.
    - Una vez se recibe la respuesta, informa al usuario mediante una alerta y actualiza la vista con la lista actualizada.
  - 5. **eliminarComentario()**
    - Recupera el ID del comentario a eliminar desde el campo idBuscar.
    - Ejecuta una solicitud DELETE al endpoint correspondiente.
    - Muestra un mensaje de confirmación al usuario y actualiza la lista.
  - 6. **mostrarSeccion(id)**
    - Función auxiliar para controlar la visibilidad de las secciones HTML.
    - Oculta todas las secciones con clase .seccion y luego muestra solo aquella cuyo id coincide con el argumento.
    - Este enfoque simula una navegación tipo SPA (Single Page Application), sin recargar la página.
  - 7. **Inicialización**
    - Al cargar la página, se ejecutan:
      - listarComentarios() para poblar inicialmente la lista de comentarios.
      - mostrarSeccion('crear') para mostrar por defecto la sección de creación.
- 

#### **Flujo de interacción:**

1. El usuario ingresa su comentario, email y calificación.
2. El navegador envía los datos al microservicio expuesto por FastAPI a través de fetch().
3. Si el comentario es negativo (calificación ≤ 2), el backend activa el patrón Observer y registra la notificación.
4. La respuesta del backend permite actualizar dinámicamente la lista de comentarios en pantalla.

#### ► **Interfaz de Escritorio**

```
python -m frontend.escritorio.escritorio
```

Gestión de Comentarios

ID	Email	Comentario	Calificación	Fecha
6	corregido	actualizado nuevamente	2	2025-06-16T13:08:25
7	adfasdf	actualizado nuevamente	1	2025-06-16T13:09:54
8	asdfa@hotmail.com	asdf	5	2025-06-16T13:16:14
12	corregido	actualizado nuevamente	1	2025-06-16T23:17:41
15	correo	comentario	4	2025-06-17T00:00:22
16	asdfa	sdaf	2	2025-06-17T00:13:15
17	abeltran@uniempresarial.e	dddd	1	2025-06-17T07:01:31
18	adsf	adsfa	1	2025-06-17T08:20:35
19	asdf	12	1	2025-06-17T08:31:59
20	1221	1212	1	2025-06-17T08:37:49

Nuevo | Editar | Eliminar | Recargar |

comentario\_service.py 18 <textarea id="texto" placeholder="Comentario"></textarea>

## frontend\escritorio\escritorio.py

```

import tkinter as tk
from tkinter import ttk, messagebox
import requests
import json

# Cargar configuración desde config.json
with open("modulos/comentarios/configuracion/config.json") as f:
    config = json.load(f)

API = config["api_base"]
ENDPOINTS = config["endpoints"]

def recargar_datos():
    for item in tree.get_children():
        tree.delete(item)
    try:
        r = requests.get(API + ENDPOINTS["read_all"])
        if r.status_code == 200:
            for c in r.json():
                tree.insert("", "end", values=(
                    c["id"], c["usuario_email"], c["texto"], c["calificacion"],
                    c.get("fecha", ""))
    except Exception as e:
        messagebox.showerror("Error", str(e))

def crear_comentario():
    dialogo_comentario("Crear nuevo comentario")

def editar_comentario():
    seleccionado = tree.focus()
    if not seleccionado:
        messagebox.showwarning("Seleccionar", "Seleccione un comentario.")
        return
    valores = tree.item(seleccionado, "values")
    dialogo_comentario("Editar comentario", valores)

def eliminar_comentario():

```

```

seleccionado = tree.focus()
if not seleccionado:
    messagebox.showwarning("Seleccionar", "Seleccione un comentario.")
    return
id_com = tree.item(seleccionado, "values")[0]
if messagebox.askyesno("Eliminar", "¿Seguro que desea eliminar este comentario?"):
    try:
        r = requests.delete(API + ENDPOINTS["delete"].replace("{id}", str(id_com)))
        if r.status_code == 200:
            recargar_datos()
            messagebox.showinfo("Éxito", "Comentario eliminado.")
        else:
            messagebox.showerror("Error", r.text)
    except Exception as e:
        messagebox.showerror("Error", str(e))

def dialogo_comentario(titulo, datos=None):
    ventana = tk.Toplevel(root)
    ventana.title(titulo)

    tk.Label(ventana, text="Email:").grid(row=0, column=0)
    email = tk.Entry(ventana)
    email.grid(row=0, column=1)

    tk.Label(ventana, text="Comentario:").grid(row=1, column=0)
    texto = tk.Entry(ventana)
    texto.grid(row=1, column=1)

    tk.Label(ventana, text="Calificación:").grid(row=2, column=0)
    calificacion = tk.Entry(ventana)
    calificacion.grid(row=2, column=1)

    if datos:
        id_com, email_val, texto_val, calif_val, _ = datos
        email.insert(0, email_val)
        texto.insert(0, texto_val)
        calificacion.insert(0, calif_val)

    def guardar():
        payload = {
            "usuario_email": email.get(),
            "texto": texto.get(),
            "calificacion": calificacion.get()
        }
        try:
            if datos:
                r = requests.put(API + ENDPOINTS["update"].replace("{id}", str(id_com)), json=payload)
            else:
                r = requests.post(API + ENDPOINTS["create"], json=payload)
            if r.status_code in [200, 201]:
                recargar_datos()
                ventana.destroy()
                messagebox.showinfo("Éxito", "Operación exitosa.")
            else:
                messagebox.showerror("Error", r.text)
        except Exception as e:
            messagebox.showerror("Error", str(e))

    tk.Button(ventana, text="Guardar", command=guardar).grid(row=3, columnspan=2)

```

```

# Ventana principal
root = tk.Tk()
root.title("Gestión de Comentarios")

# Tabla
cols = ("ID", "Email", "Comentario", "Calificación", "Fecha")
tree = ttk.Treeview(root, columns=cols, show="headings")
for col in cols:
    tree.heading(col, text=col)
    tree.column(col, anchor="center", width=150)
tree.pack(fill="both", expand=True, padx=10, pady=10)

# Botones CRUD
botonera = tk.Frame(root)
botonera.pack(pady=10)

tk.Button(botonera, text="Nuevo", command=crear_comentario).pack(side="left", padx=5)
tk.Button(botonera, text="Editar", command=editar_comentario).pack(side="left", padx=5)
tk.Button(botonera, text="Eliminar", command=eliminar_comentario).pack(side="left",
    padx=5)
tk.Button(botonera, text="Recargar", command=recargar_datos).pack(side="left", padx=5)

recargar_datos()
root.mainloop()

```

### Carga de configuración externa

- Se importa json para leer el archivo config.json, desde el cual se extraen:
  - API: la URL base del microservicio.
  - ENDPOINTS: diccionario con las rutas para operaciones CRUD.
- Este enfoque respeta el principio de configuración externa, promoviendo la flexibilidad del entorno.

### Función recargar\_datos()

- Limpia el contenido actual del Treeview que muestra los comentarios.
- Realiza una solicitud GET al endpoint de lectura general (read\_all) usando requests.get().
- Si la respuesta es exitosa (status\_code == 200), inserta en la tabla cada comentario como una fila con ID, email, texto, calificación y fecha.
- Utiliza el widget ttk.Treeview, adecuado para representar listas tabulares.

### Función crear\_comentario()

- Invoca dialogo\_comentario() sin argumentos, lo cual genera una ventana emergente (Toplevel) para crear un nuevo comentario.

### Función editar\_comentario()

- Verifica si hay una fila seleccionada en la tabla.
- Extrae los valores del comentario seleccionado y los pasa como parámetro a dialogo\_comentario(), permitiendo su edición.

### Función eliminar\_comentario()

- Obtiene el ID del comentario seleccionado y solicita confirmación al usuario mediante un cuadro de diálogo.
- Si se confirma, se hace una solicitud DELETE al endpoint correspondiente y se recarga la tabla con los datos actualizados.

### Función dialogo\_comentario()

- Crea una ventana secundaria (Toplevel) que actúa como formulario para crear o editar comentarios.
- Carga los datos existentes (si aplica) en campos de entrada (Entry).
- Internamente define guardar(), una función que construye el payload con los valores de los campos y envía una solicitud POST (crear) o PUT (actualizar), según el caso.
- Muestra mensajes de éxito o error usando messagebox.

### Interfaz principal (root)

- Se crea una ventana raíz con Tk() y se le asigna el título *Gestión de Comentarios*.
- Se define la tabla de visualización (Treeview) con columnas para ID, Email, Comentario, Calificación y Fecha.
- Se insertan botones para operaciones CRUD y recarga:
  - **Nuevo** → llama a crear\_comentario()
  - **Editar** → llama a editar\_comentario()
  - **Eliminar** → llama a eliminar\_comentario()
  - **Recargar** → vuelve a consultar la API

### Ejecución del ciclo principal

- recargar\_datos() se ejecuta al inicio para cargar los comentarios existentes.
- root.mainloop() mantiene la ventana activa esperando eventos del usuario.

### Principios aplicados:

- **Separación de lógica y presentación**: la lógica de negocio está en funciones independientes del diseño de la UI.
- **Consumo de servicios REST desde cliente Python**: usando requests, el cliente Tkinter se comunica con los microservicios.
- **Modularidad y reusabilidad**: las funciones CRUD están separadas, facilitando mantenimiento y extensiones.
- **UX intuitiva**: el uso de cuadros de diálogo (messagebox, Toplevel) mejora la interacción usuario-aplicación.

### ► Interfaz de Consola

```
python -m frontend.consola.consola
```

## frontend\consola\consola.py

```
import requests
import json

# Cargar configuración desde config.json
with open("modulos/comentarios/configuracion/config.json") as f:
    config = json.load(f)

API = config["api_base"]
ENDPOINTS = config["endpoints"]

def menu():
    print("\n===== Menú Principal =====")
    print("1. Ingresar nuevo comentario")
    print("2. Listar comentarios")
    print("3. Obtener comentario por ID")
    print("4. Actualizar comentario")
```

```

        print("5. Eliminar comentario")
        print("6. Salir")

def ingresar():
    texto = input("Texto: ")
    email = input("Email: ")
    calificacion = input("Calificación (1-5): ")
    r = requests.post(f"{API}{ENDPOINTS['create']}", json={
        "texto": texto,
        "usuario_email": email,
        "calificacion": int(calificacion)
    })
    print(r.json())

def listar():
    r = requests.get(f"{API}{ENDPOINTS['read_all']}")
    for c in r.json():
        print(c)

def obtener():
    id = input("ID del comentario: ")
    url = ENDPOINTS["read_one"].replace("{id}", id)
    r = requests.get(f"{API}{url}")
    print(r.json() if r.status_code == 200 else "Comentario no encontrado.")

def actualizar():
    id = input("ID a actualizar: ")
    texto = input("Nuevo texto: ")
    email = input("Nuevo email: ")
    calificacion = input("Nueva calificación: ")
    url = ENDPOINTS["update"].replace("{id}", id)
    r = requests.put(f"{API}{url}", json={
        "texto": texto,
        "usuario_email": email,
        "calificacion": int(calificacion)
    })
    print(r.json())

def eliminar():
    id = input("ID a eliminar: ")
    url = ENDPOINTS["delete"].replace("{id}", id)
    r = requests.delete(f"{API}{url}")
    print(r.json())

if __name__ == "__main__":
    while True:
        menu()
        opcion = input("Seleccione una opción: ")
        if opcion == "1": ingresar()
        elif opcion == "2": listar()
        elif opcion == "3": obtener()
        elif opcion == "4": actualizar()
        elif opcion == "5": eliminar()
        elif opcion == "6": break
        else: print("Opción inválida.")

```

## 1. Carga de configuración externa

- Se utiliza el módulo json para leer el archivo config.json, extrayendo:
  - API: la URL base del microservicio.

- ENDPOINTS: las rutas relativas para cada operación CRUD.
  - Esto permite desacoplar el código fuente de los detalles de configuración, promoviendo la reutilización en diferentes entornos.
2. **Menú principal (menu())**
- Imprime un menú textual con opciones numeradas para realizar operaciones CRUD:
    1. Crear comentario
    2. Listar todos
    3. Buscar por ID
    4. Actualizar
    5. Eliminar
    6. Salir
  - Representa la interfaz de usuario de tipo consola, ideal para pruebas rápidas o ambientes de línea de comandos.
3. **ingresar() – Crear comentario**
- Solicita al usuario el texto, correo y calificación.
  - Realiza una solicitud POST al endpoint de creación con el cuerpo en formato JSON.
  - Imprime la respuesta del servidor, que puede incluir un mensaje de éxito o error.
4. **listar() – Leer todos los comentarios**
- Ejecuta una solicitud GET al endpoint de lectura general.
  - Itera sobre la lista de comentarios recibida y los imprime uno por uno como diccionarios.
5. **obtener() – Leer por ID**
- Solicita al usuario el ID del comentario que desea consultar.
  - Reemplaza {id} en la URL correspondiente y realiza una solicitud GET.
  - Muestra el contenido si la respuesta es exitosa (status\_code == 200), o informa si no se encontró el comentario.
6. **actualizar() – Modificar comentario existente**
- Pide al usuario el ID del comentario a modificar, así como los nuevos valores.
  - Realiza una solicitud PUT al backend con los datos actualizados en JSON.
  - Muestra la respuesta del servidor, generalmente un mensaje de confirmación.
7. **eliminar() – Borrar comentario**
- Solicita el ID del comentario a eliminar.
  - Ejecuta una solicitud DELETE y muestra el resultado devuelto por el microservicio.
8. **Ejecución principal (\_\_main\_\_)**
- Ejecuta un ciclo while infinito que muestra el menú y espera una entrada del usuario.
  - Dependiendo de la opción seleccionada, invoca la función correspondiente.
  - La opción "6" rompe el ciclo, finalizando el programa.
  - Si la entrada no corresponde a ninguna opción válida, muestra un mensaje de error.

---

#### Principios aplicados:

- **Diseño modular:** cada operación está encapsulada en una función específica.

- **Separación de lógica y configuración:** uso de archivo config.json para endpoints y URL base.
- **Interacción RESTful:** consumo de servicios web mediante requests para cada operación CRUD.
- **Simplicidad y portabilidad:** al ser una interfaz de línea de comandos, puede ejecutarse en cualquier entorno sin dependencias gráficas.
- **Buena práctica de entrada/salida:** solicita confirmación y muestra retroalimentación para cada acción.

## gateway\main.py

The screenshot shows a code editor with the main.py file open. The project structure on the left includes a LAB-PROYECTOFINAL folder with subfolders like frontend, gateway, modulos, and logs. The gateway folder contains files such as \_\_pycache\_\_, main.py, and comentario\_service.py. The main.py file itself imports FastAPI and CORSMiddleware from fastapi.middleware.cors, and it includes a router from the modulo comentarios. The code is as follows:

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from modulos.comentarios.logica.comentario_service import router as comentarios_router

app = FastAPI(title="API Gateway")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"])

app.include_router(comentarios_router, prefix="/comentarios")
```

### 1. Propósito

- Actúa como el **punto de entrada principal** de la arquitectura basada en microservicios.
- Implementa el patrón **API Gateway**, que centraliza el enrutamiento hacia los distintos microservicios.

### 2. Componentes principales

- FastAPI(title="API Gateway"): instancia principal de la aplicación web.
- CORS (Cross-Origin Resource Sharing): permite que los clientes (por ejemplo, la interfaz web) accedan a la API desde cualquier origen.
- comentarios\_router: se importa desde el microservicio de comentarios y se monta en el endpoint /comentarios.

### 3. Middleware configurado

- CORSMiddleware: habilita solicitudes de cualquier dominio (allow\_origins=["\*"]) y permite compartir credenciales, métodos y cabeceras.
- Esto es fundamental para permitir que aplicaciones web o móviles consuman la API sin restricciones de seguridad del navegador.

### 4. Ruta incluida

- app.include\_router(comentarios\_router, prefix="/comentarios"): integra todas las rutas del microservicio de comentarios bajo el prefijo /comentarios.

### Ventajas del diseño

- **Centralización**: todos los servicios pueden ser registrados aquí, bajo distintos prefijos (/comentarios, /usuarios, etc.).
- **Escalabilidad**: se pueden agregar más microservicios sin cambiar la lógica interna de los existentes.
- **Flexibilidad de acceso**: gracias al CORS, se facilita la integración con interfaces distribuidas (web, móvil, escritorio).

```
python -m uvicorn gateway.main:app --reload
```

```
PS C:\Users\Adam\Mi unidad\LABS\Principios POO\Lab-ProyectoFinal> python -m uvicorn gateway.main:app --reload
INFO: Will watch for changes in these directories: ['C:\\\\Users\\\\Adam \\\\Mi unidad\\\\LABS\\\\Principios POO\\\\Lab-ProyectoFinal']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [15432] using StatReload
INFO: Started server process [9544]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Este comando se utiliza para iniciar el servidor de desarrollo del sistema utilizando **Uvicorn**, un servidor ASGI de alto rendimiento compatible con FastAPI.

Especificamente:

1. **python -m uvicorn**

Ejecuta el módulo uvicorn como intérprete principal utilizando el ejecutable de Python.

2. **gateway.main:app**

Indica la ruta al punto de entrada de la aplicación FastAPI:

- o gateway es el nombre de la carpeta.
- o main es el archivo (main.py).
- o app es la instancia de FastAPI definida en dicho archivo.

3. **--reload**

Activa el modo de recarga automática. El servidor se reinicia cada vez que se detecta un cambio en los archivos del proyecto, ideal para entornos de desarrollo.

---

### 💡 Beneficios del uso

- Permite iniciar la API Gateway, centralizando el acceso a los microservicios definidos.
- Habilita un entorno de desarrollo ágil gracias al modo --reload.
- Facilita pruebas rápidas de cambios en las rutas, configuración o lógica de negocio.