Nombre: Monica Pardo

ID de usuario para API: M9JRA3

El archivo .env contiene la variable USER_ID con valor M9JRA3

Sistema de Barrido de Llamadas

Este proyecto implementa un sistema para:

- Registrar llamadas iniciales desde una API.
- Realizar un proceso de "barrido" para mejorar registros con categoría *bad* hasta que todos sean *medium o good.*
- Generar estadísticas y resultados del proceso.

Tecnologías utilizadas

- Python 3.10+
- SQLAlchemy (ORM para conexión a base de datos)
- Pytest (pruebas automatizadas)
- PostgreSQL o MySQL (base de datos relacional)
- Requests (para consumo de APIs externas)
- Flask (para exponer endpoints CRUD)

Estructura del proyecto

prueba-tecnica/

|

├─ app/

| ├─ models.py # Definición de tablas y modelos

| ├─ repositories/ # Acceso a datos

│ ├─ services/ # Lógica de negocio

│ └─ controllers/ # Controladores principales

│

├─ scripts/ # Scripts ejecutables

├─ tests/ # Pruebas automatizadas

├─ requirements.txt # Dependencias

├─ run.py # Punto de entrada principal

└─ README.md # Documentación

Instalación y ejecución

1. Clonar repositorio

git clone https://github.com/usuario/prueba-tecnica.git

cd prueba-tecnica

2. Crear entorno virtual

Linux / Mac

python -m venv venv

source venv/bin/activate

Windows

python -m venv venv

venv\Scripts\activate

```
C:\Users\monica alejandra\OneDrive\prueba-tecnica>python -m venv venv

C:\Users\monica alejandra\OneDrive\prueba-tecnica>venv\Scripts\activate

(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>pip install python-dotenv
Requirement already satisfied: python-dotenv in c:\users\monica alejandra\onedrive\prueba-tecnica\venv\lib\site-packages
 (1.1.1)
```

3. Instalar dependencias

pip install -r requirements.txt

4. Crear tablas y cargar datos iniciales

python scripts/create_tables.py

python scripts/initial_load.py

```
(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>python -m scripts.create_tables
Tablas creadas correctamente.
```

```
(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>python -m scripts.initial_load
Iniciando carga inicial de 100 llamadas...
1/100 -> id=1 cat=bad value=13
2/100 -> id=2 cat=good value=92
3/100 -> id=3 cat=bad value=42
4/100 -> id=4 cat=bad value=19
5/100 -> id=5 cat=good value=88
6/100 -> id=6 cat=bad value=3
7/100 -> id=7 cat=good value=98
8/100 -> id=8 cat=bad value=38
9/100 -> id=9 cat=medium value=83
10/100 -> id=10 cat=bad value=25
11/100 -> id=11 cat=good value=87
12/100 -> id=12 cat=medium value=73
```

5. Ejecutar el barrido

python scripts/run_barrido.py

```
(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>python -m scripts.run_barrido
Resultado del barrido: {'sweeps': 9}

(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>python -m scripts.run_barrido
Resultado del barrido: {'sweeps': 0}
```

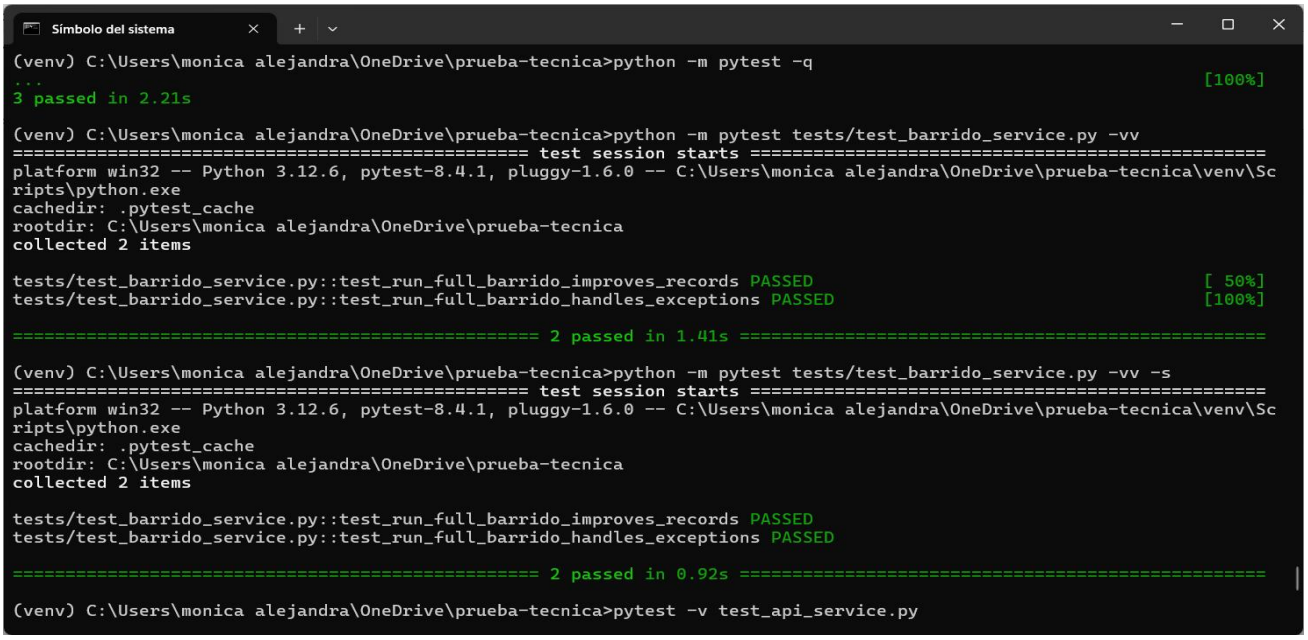
6. Consultar estado actual

python scripts/check_status.py

```
(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>python -m scripts.check_status
=== ESTADO ===
Total registros: 300
Distribución por categoría:
  - medium: 188
  - good: 112
Llamadas totales (SUM attempts): 718
Barridos realizados: 19
```

7. Ejecutar pruebas

pytest -v



Ejecución en Postman

Levantar la API

python run.py

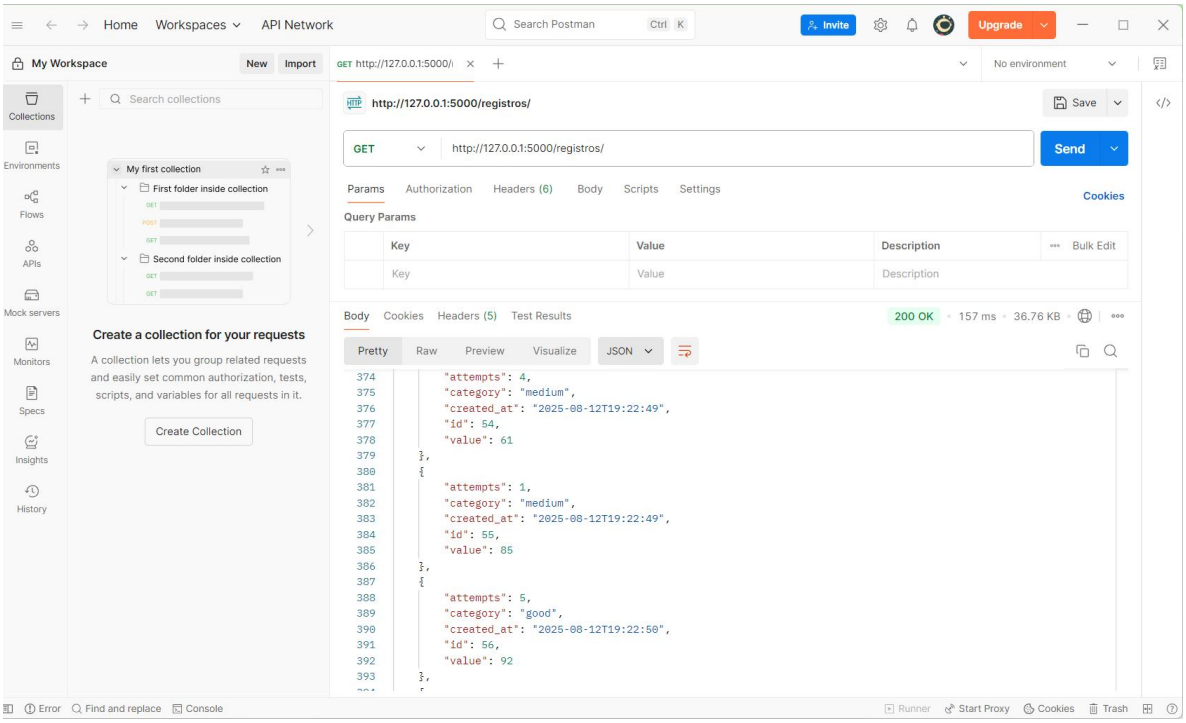
```
(venv) C:\Users\monica alejandra\OneDrive\prueba-tecnica>python run.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 570-264-442
```

Listar registros (GET)

URL: http://127.0.0.1:5000/registros/

Método: GET

URL: http://127.0.0.1:5000/registros/



Crear registro (POST)

URL: http://127.0.0.1:5000/registros/

Método: POST

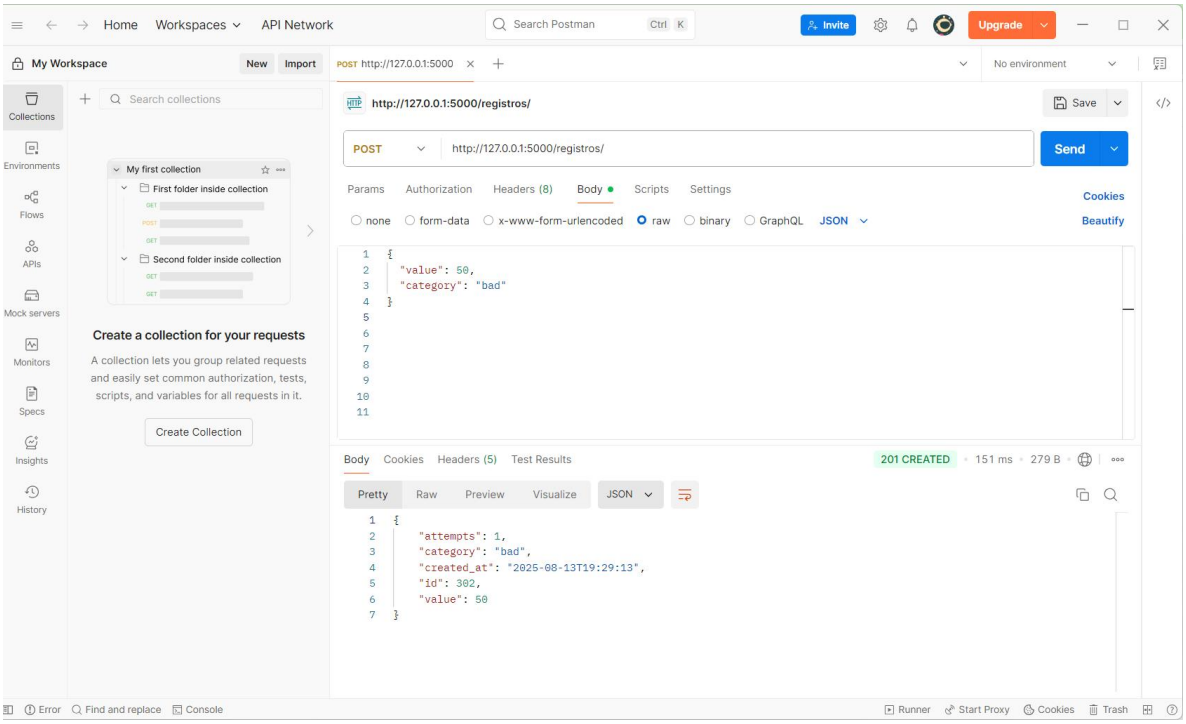
Body (JSON):

{

"value": 50,

"category": "bad"

}



Actualizar registro (PUT)

URL: http://127.0.0.1:5000/registros/6

Método: PUT

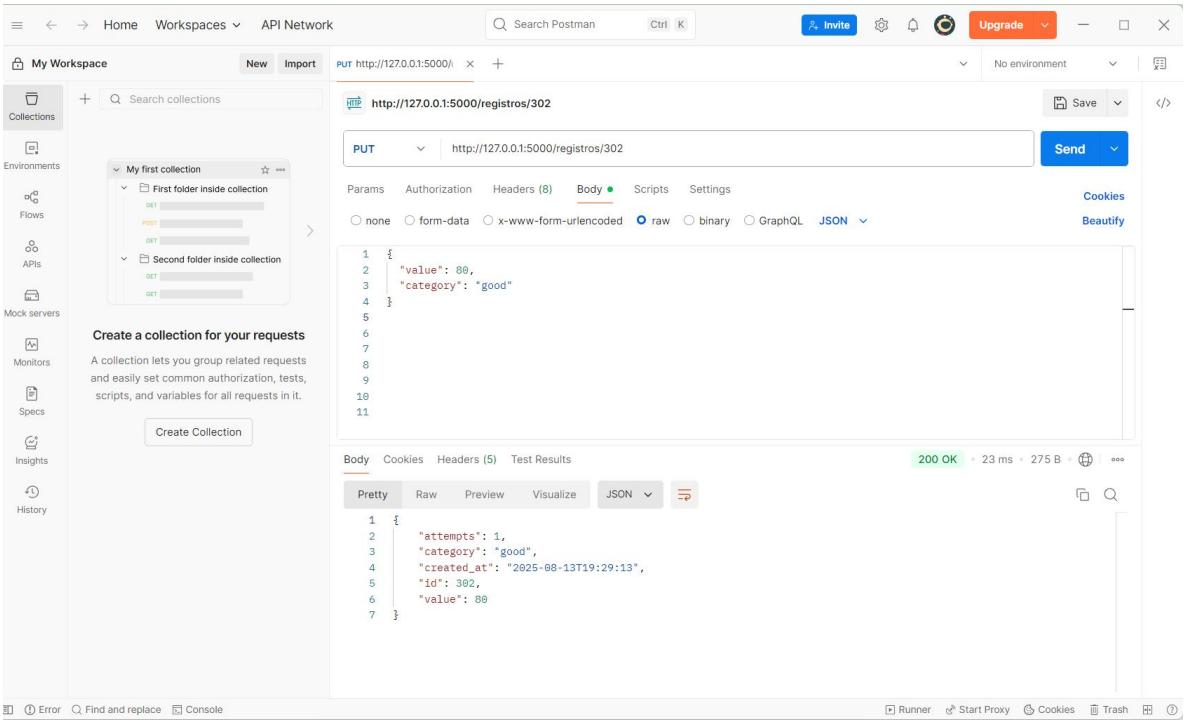
Body (JSON):

{

"value": 80,

"category": "good"

}



Eliminar registro (DELETE)

URL: http://127.0.0.1:5000/registros/6

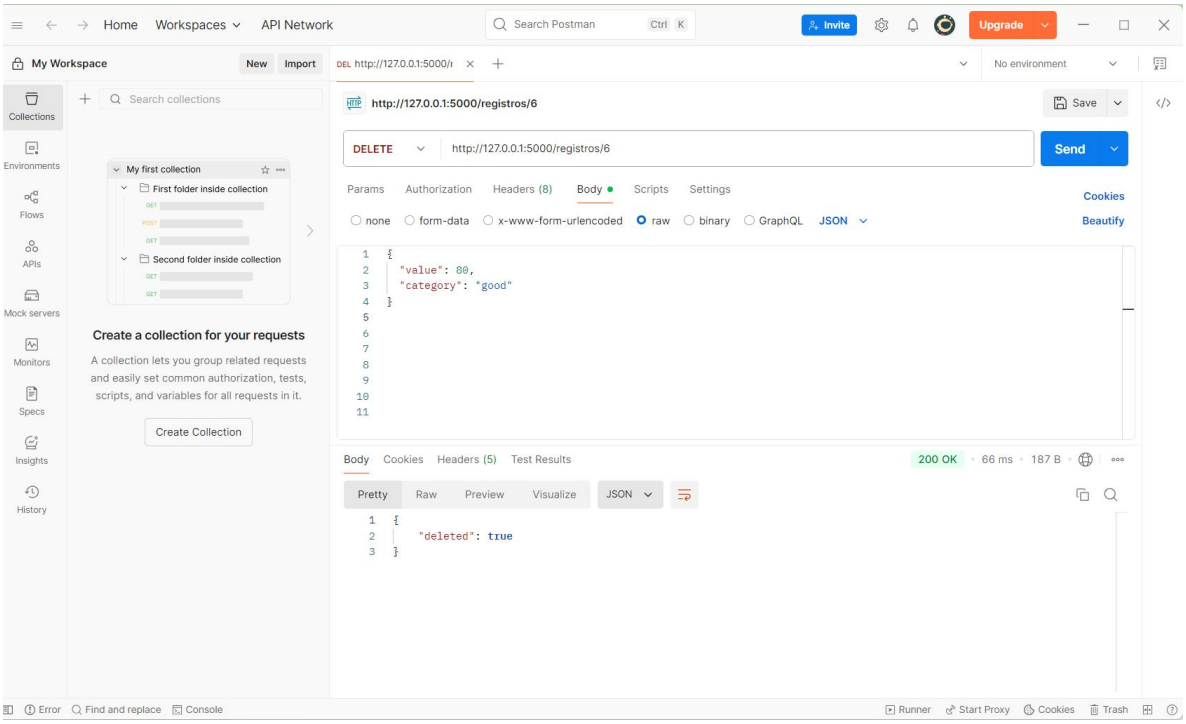
Método: DELETE

Respuesta esperada:

{

"message": "True"

}



Resultados del barrido

Número de llamadas iniciales: 100

Número de barridos realizados: 19

Total de registros finales: 300

Total de intentos (SUM attempts): 718

Distribución final por categoría:

good: 112

medium: 188

bad: 0

SQL consultas

Arquitectura y principios SOLID

Single Responsibility: Cada servicio, repositorio y controlador tiene su propia responsabilidad.

Open/Closed: Fácil de extender sin modificar código base.

Liskov Substitution: Interfaces claras para reemplazar implementaciones.

Interface Segregation: Clases separadas para cada acción concreta.

Dependency Inversion: Inyección de dependencias para desacoplar módulos.

Consultas SQL útiles para el reporte final

Total de registros

sql

```
SELECT COUNT(*) FROM registros;
```

Distribución por categoría

sql

```
SELECT category, COUNT(*) AS cantidad
FROM registros
GROUP BY category;
```

Total de llamadas (sumando attempts)

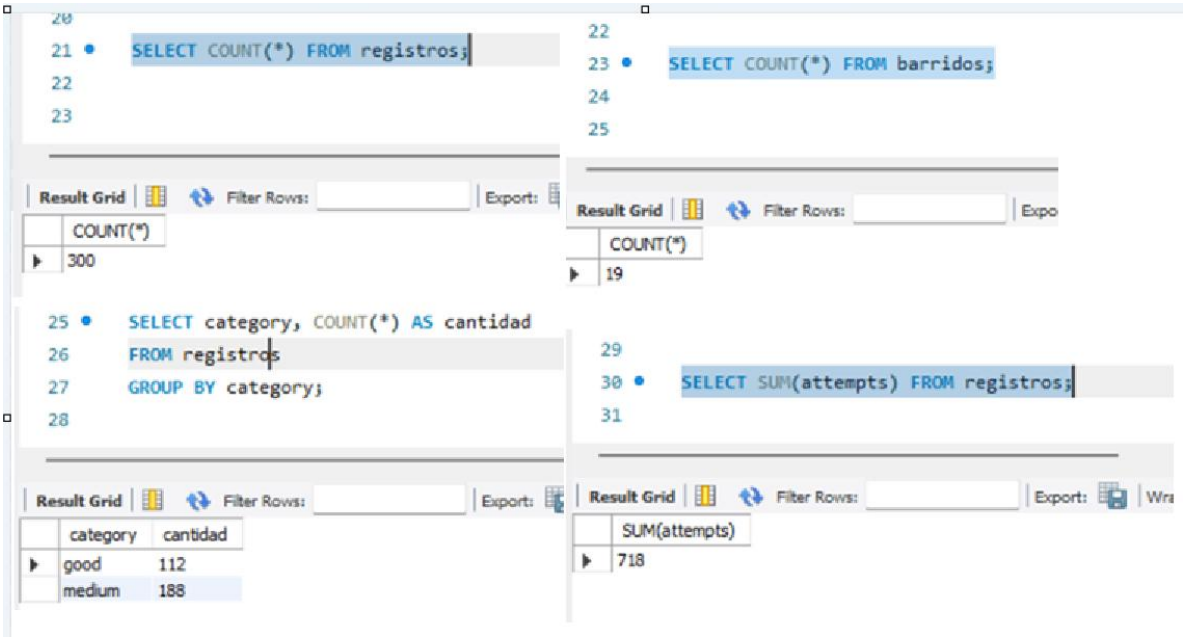
sql

```
SELECT SUM(attempts) AS llamadas_totales
FROM registros;
```

Barridos realizados

sql

```
SELECT COUNT(*) AS num_barridos
FROM barridos;
```



Esquema de tablas

sql

CREATE TABLE registros (

id INT AUTO_INCREMENT PRIMARY KEY,

value INT NOT NULL,

attempts INT NOT NULL DEFAULT 0,

category ENUM('bad', 'medium', 'good') NOT NULL,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP

);

CREATE TABLE barridos (

id INT AUTO_INCREMENT PRIMARY KEY,

sweep_number INT NOT NULL,

records_checked INT NOT NULL,

records_improved INT NOT NULL,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);

Manejo de errores, reintentos y anti-ráfagas

Cliente HTTP (ApiService)

- `**timeout**`: 6s por solicitud (evita que una llamada cuelgue el proceso).
- `**max_retries**`: 3 intentos por llamada cuando hay errores transitorios.
- `**backoff_factor**`: 0.5 (esperas crecientes: 0.5s, 1.0s, 2.0s…).
- **Validación de respuesta:** conversión segura de `value` a `int` y chequeo de `category` (`bad|medium|good`).
- ****Anti-ráfagas****: `min_delay=0.12s` entre llamadas → ~8.3 req/s máx.

Barrido (BarridoService)

- Repite ciclos mientras existan `bad` (hasta `max_sweeps`).
- Cada intento suma en `attempts` aun si la llamada falla (trazabilidad).
- Si la nueva categoría es `medium|good`, actualiza el registro; si no, lo deja igual.
- Si el API falla en un registro, se registra el intento y el barrido continúa (no cae todo el proceso).

Controladores (CRUD)

- Responden JSON coherente.
- Errores comunes:
- `400` (datos inválidos): falta `value` o `category`.
- `404` (no encontrado): id inexistente.
- `500` (error interno): error no controlado.

Reproducibilidad

- `scripts/check_status.py` imprime métricas (total, distribución, SUM(attempts), barridos).
- Consultas SQL equivalentes están documentadas.

Resumen final

El sistema logró barrer todas las llamadas con categoría bad en 19 iteraciones, quedando 0 registros en esa categoría

y un total de 718 intentos acumulados. Esto garantiza que todos los registros finales están en estado medium o good, cumpliendo

con los objetivos planteados.