INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

"Aplicaciones Distribuidas"

Actividad 1

Creación de un API

DARIO JAVIER MORALES CAIZA

NRC: 3877

Integrantes:

- Moisés Benjamín Socasi Gualichico
- Adrián Isaee Simbaña Moreira
- Marlon Pavel Torres Chávez

Periodo Académico

S-I NOVIEMBRE 24 – MAYO 25

Contenido

Introducción

La gestión eficiente de información es fundamental para el éxito de cualquier organización. Las APIs (Interfaces de Programación de Aplicaciones) permiten la comunicación entre diferentes sistemas, facilitando la integración de servicios y la automatización de procesos. Este informe presenta el desarrollo de una API para la gestión de datos de productos, utilizando Python como lenguaje de programación y MySQL como sistema de gestión de bases de datos. La API permite realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sobre una base de datos de productos, optimizando y mejorando la accesibilidad de la información para diferentes aplicaciones.

Objetivos

Principal

Desarrollar una API utilizando Python y MySQL para la gestión eficiente de productos, que permita realizar operaciones CRUD de forma segura y efectiva, proporcionando una plataforma para la administración centralizada de datos

Específicos

- Implementar una base de datos en MySQL que almacene la información relevante de los productos, incluyendo detalles como nombre, descripción, precio.
- Crear una API con Python usando un framework adecuado (Flask) para facilitar la interacción con la base de datos.
- Desarrollar y documentar endpoints que permitan la gestión completa de productos (operaciones CRUD) a través de métodos HTTP.
- Realizar pruebas exhaustivas de la API para asegurar su funcionalidad y rendimiento, utilizando herramientas como Postman.

Justificación

La creación de una API usando lenguajes como Python y bases de datos como MySQL son una combinación ideal para la creación de APIs debido a su simplicidad y eficiencia. Python es conocido por su sintaxis clara y su amplia gama de bibliotecas, lo que facilita el desarrollo rápido de aplicaciones web mediante frameworks como Flask o Django. Además, cuenta con herramientas sólidas para la validación de datos y la gestión de errores. MySQL, por otro lado, es una base de datos relacional robusta, escalable y ampliamente utilizada, que ofrece una alta eficiencia en la manipulación y consulta de datos. La combinación de Python y MySQL permite crear aplicaciones backend que son tanto fáciles de mantener como altamente confiables.

Desarrollo

Estructura del API

app.py

El archivo app.py es el núcleo de nuestra API RESTful creada con Flask, que gestiona productos almacenados en una base de datos MySQL. Definiendo dos rutas: una para listar productos (GET /productos) y otra para agregar nuevos productos (POST /productos). Las conexiones a la base de datos se manejan de manera eficiente, abriéndose y cerrándose en cada operación para garantizar estabilidad y seguridad en la API.

```
from flask import Flask, request, jsonify
from config import create_app, get_db_connection

app = create_app()

@app.route('/productos', methods=['GET'])
def listar_productos():
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM productos")
    productos = cursor.fetchall()
    cursor.close()
    conn.close()
    return jsonify(productos), 200

@app.route('/productos', methods=['POST'])
def agregar_producto():
    data = request.json
    if not data or 'nombre' not in data or 'precio' not in data:
        return jsonify({'error': 'Datos faltantes'}), 400

    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)",
(data['nombre'], data['precio']))
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'mensaje': 'Producto agregado'}), 201

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

config.py

Establecemos la configuración básica para la aplicación Flask y la conexión a la base de datos MySQL. Contiene dos funciones principales, la función create_app() inicializa la instancia de la aplicación Flask. Por otro lado, get_db_connection() es una función que gestiona la conexión a la base de datos MySQL utilizando las variables de entorno, manteniendo los datos sensibles seguros al no almacenarlos directamente en el código fuente.

```
import os
import mysql.connector
from flask import Flask

def create_app():
    app = Flask(__name__)
    return app

def get_db_connection():
    return mysql.connector.connect(
        host=os.getenv('MYSQL_HOST'),
        user=os.getenv('MYSQL_USER'),
        password=os.getenv('MYSQL_PASSWORD'),
        database=os.getenv('MYSQL_DB'),
        port=os.getenv('MYSQL_PORT')
    )
```

docker-compose.yml

En este archivo se configura dos servicios esenciales para el despliegue de la API con Docker: la base de datos MySQL y la API en Flask. El servicio db utiliza una imagen de MySQL 8.0 y define variables de entorno para la configuración de la base de datos, incluyendo la contraseña de root, el nombre de la base de datos (productos_db), y las credenciales de un usuario específico. La base de datos expone el puerto 3306 (mapeado al 3307 en el host) y almacena sus datos en un volumen llamado db_data, que asegura la persistencia de la información incluso si el contenedor se reinicia.

```
services:
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: productos_db
      MYSQL_USER: user
      MYSQL_PASSWORD: password
    ports:
      - '3307:3306'
    volumes:
      - db_data:/var/lib/mysql

  api:
    build: .
    ports:
      - '5000:5000'
    environment:
      - MYSQL_USER=user
      - MYSQL_PASSWORD=password
      - MYSQL_DB=productos_db
      - MYSQL_HOST=db
      - MYSQL_PORT=3306
    depends_on:
      - db
```

```
    restart: always

volumes:
  db_data:
```

Dockerfile

En el dockerfile configuramos un contenedor con Python 3.9, copiamos el código de la aplicación Flask e instalamos las dependencias necesarias. Exponemos el puerto 5000 para la accesibilidad y ejecuta el archivo app.py para iniciar la aplicación. Esto nos facilita la ejecución de la API en un entorno aislado y eficiente.

```
# Utilizar una imagen base de Python
FROM python:3.9-slim

# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar el código fuente al contenedor
COPY . .

# Instalar dependencias directamente usando pip
RUN pip install --no-cache-dir flask flask-restful mysql-connector-python pandas python-dateutil tzdata python-dotenv

# Exponer el puerto en el que Flask se ejecutará
EXPOSE 5000

# Comando para ejecutar la aplicación
CMD ["python", "app.py"]
```

Models.py

En este archivo definimos el modelo de nuestra base datos, lo que nos permite ejecutar las consultas, gestionando la base de datos de manera eficiente.

```
from config import mysql

def create_product(data):
    conn = mysql.connection
    cursor = conn.cursor()
    query = "INSERT INTO productos (nombre, precio) VALUES (%s, %s)"
    cursor.execute(query, (data['nombre'], data['precio']))
    conn.commit()
    cursor.close()

def get_all_products():
    cursor = mysql.connection.cursor()
    cursor.execute("SELECT * FROM productos")
    products = cursor.fetchall()
    cursor.close()
    return products
```

requirements.txt

Este archive txt contiene todos los requisitos necesarios para que nuestra API funcione correctamente y se integre de manera adecuada.

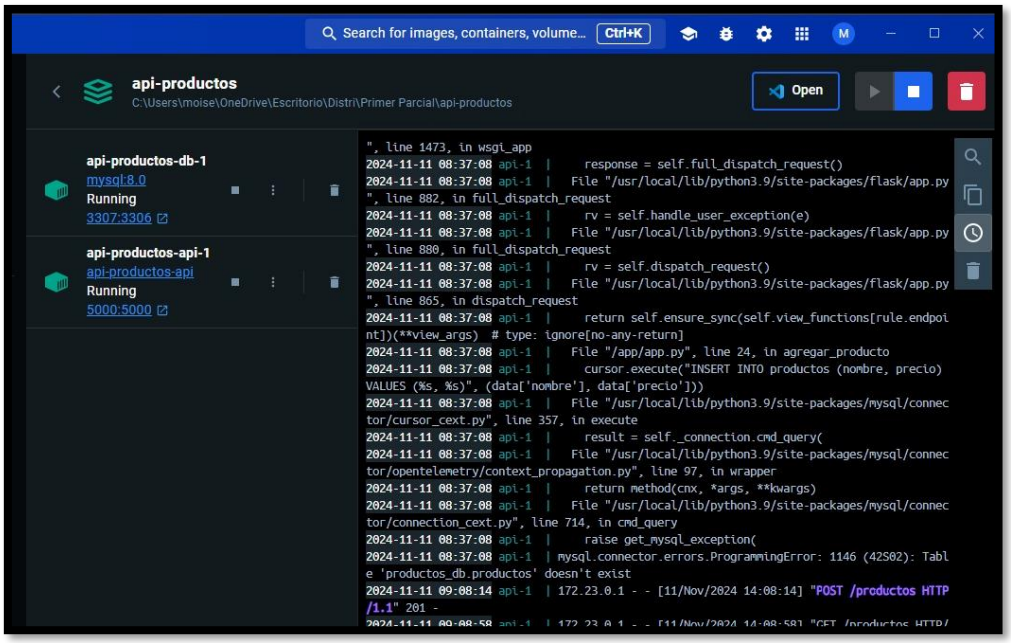
```
flask
flask-mysql-connector
flask-restful
mysql-connector-python
pandas
python-dateutil
tzdata
```

Ejecución de la API

Construir las imágenes y levantar los contenedores

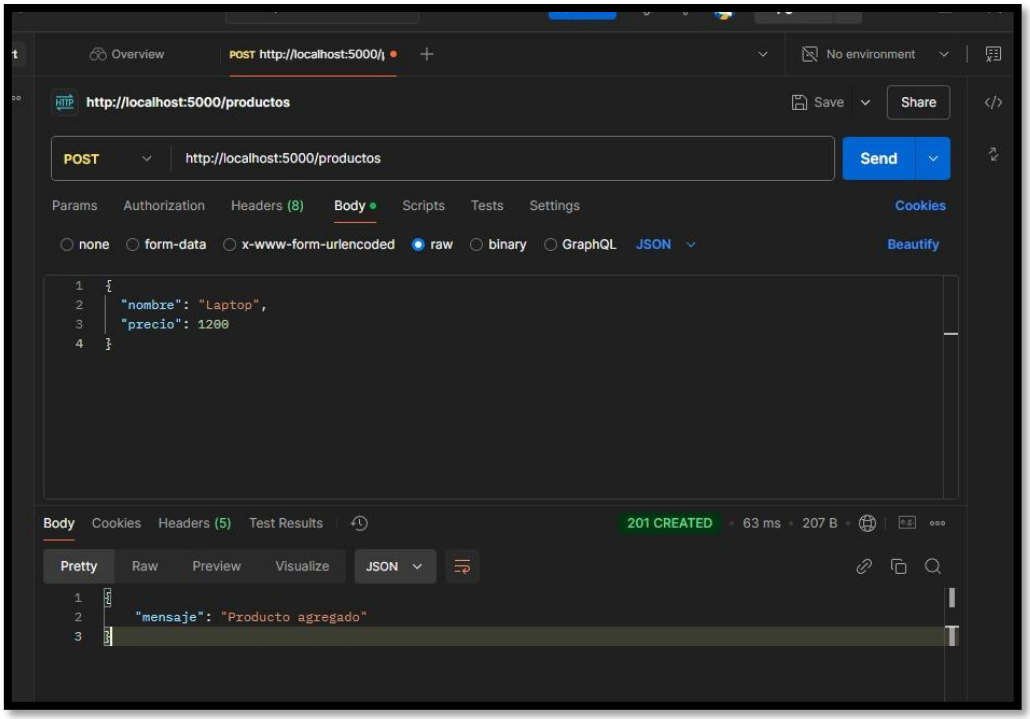
Ejecutamos el siguiente comando para construir y levantar los contenedores

```
docker-compose up --build
```



Pruebas usando postman

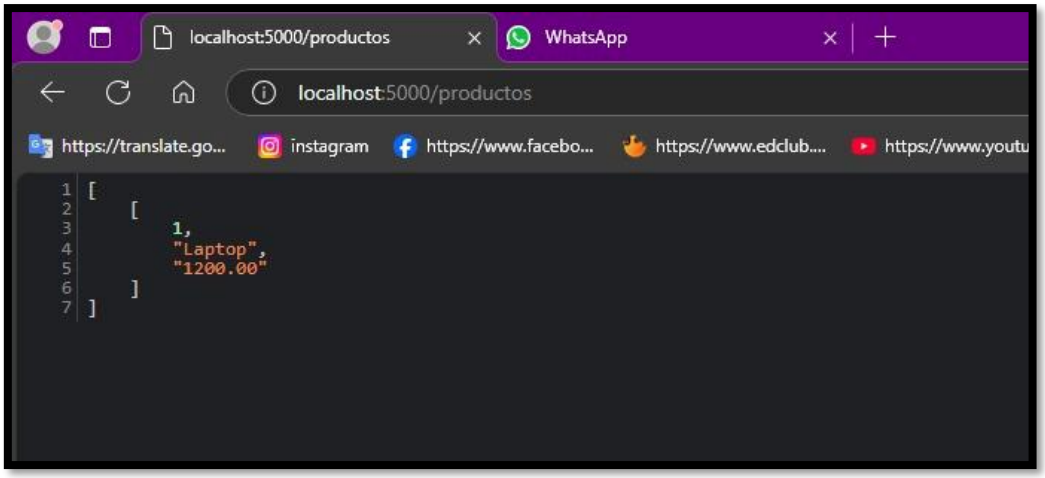
Realizamos el ingreso de productos usando herramientas como postman, que nos ayudan con peticiones POST y GET.



Acceder a la API

Una vez que los contenedores estén en funcionamiento, la API será accesible en el puerto 5000 en nuestra máquina local.

http://localhost:5000/productos



Conclusiones

- La creación de una API utilizando Python y MySQL demuestra ser una solución eficiente y escalable para gestionar productos en aplicaciones web. Gracias a la flexibilidad de Python y la robustez de MySQL, la API permite manejar operaciones CRUD básicas (crear y obtener productos) de forma sencilla y

confiable. Además, la integración de Docker facilita el proceso de despliegue, asegurando que la aplicación y su base de datos estén correctamente aisladas y configuradas en entornos reproducibles.

- Esta arquitectura basada en contenedores mejora la portabilidad y la gestión de dependencias, reduciendo el riesgo de problemas en diferentes entornos de desarrollo y producción. Sin embargo, es esencial considerar medidas de seguridad como la validación de datos y la gestión adecuada de contraseñas para garantizar la integridad de la aplicación y la protección de los datos sensibles.

Recomendaciones

- Es recomendable implementar medidas adicionales de seguridad, como el uso de HTTPS para las comunicaciones y la implementación de autenticación (por ejemplo, JWT) para controlar el acceso a la API.

- A medida que la aplicación crezca, sería útil revisar y optimizar las consultas a la base de datos, especialmente si se gestionan grandes volúmenes de datos. Considerar el uso de índices y paginación podría mejorar la eficiencia de las consultas.

- Para aplicaciones de mayor tamaño, considerar la posibilidad de dividir la arquitectura en microservicios y explorar el uso de bases de datos NoSQL o herramientas de cacheo como Redis podría ser una opción a largo plazo para mejorar el rendimiento y la escalabilidad.

Bibliografía

WebSphere Application Server Network Deployment Liberty. (2023). https://www.ibm.com/docs/es/was-liberty/nd?topic=collectives-deploying-docker-containers-using-deployment-rest-apis

Crisemcon. (2021). Guía Práctica de Contenedores: Contenerizando nuestra API con Docker y Docker Compose (2/3). DEV Community. https://dev.to/crisemcon/guia-practica-de-contenedores-contenerizando-nuestra-api-con-docker-y-docker-compose-2-3-2eb1

BetaZetaDev. (2022). Creando una API REST con Express y Docker. https://betazeta.dev/es/blog/docker-express-api-deploy/