Nombre de la práctica	Practica 4: API			No.	1
Asignatura:	Taller de Base de datos	Carrera:	Ingeniería en Sistemas Computacionales	Duración de la práctica (Hrs)	

NOMBRE DEL ALUMNO: Vanesa Hernández Martínez
GRUPO: 3501

II. Lugar de realización de la práctica (laboratorio, taller, aula u otro):
Actividades en aula de clases y en equipo personal

III. Material empleado:

- Laptop
- Navicat
- Docker

App.py

app > app.py > get_data

```python
# Creación del servidor
app = Flask(__name__)


app.config["MYSQL_HOST"]="database"
app.config["MYSQL_USER"]="vane"
app.config["MYSQL_PASSWORD"]="C413b"
app.config["MYSQL_DB"]="prueba"

mysql= MySQL(app)

@app.route("/", methods=["GET"])
def get_data():
    """
    Returns all data from the database
    """
    try:
        #creamos una conexion
        conexion = mysql.connection.cursor()
        conexion.execute("SELECT * FROM user")
        users = conexion.fetchall()
        result = []
        for user in users:
            result.append({
                "id_user":user[0],
                "username":user[1]
            })
        conexion.close()
        return jsonify(result)
    except Exception as e:
        return f"Error {str(e)}"


@app.route("/<int:id>")
def get_one_by_id(id):
    """
    Returns a data from the database by id
    """
    try:
        #creamos una conexion
        conexion = mysql.connection.cursor()
        query = "SELECT * FROM user WHERE id_user= %s"
        conexion.execute(query,(id,))
        user = conexion.fetchone()
        conexion.close()

        if user is None:
            return jsonify({"err":"User not found"}),404
        result = {
            "id_user": user[0],
            "username": user[1]
        }
        return jsonify(result)
    except Exception as e:
```

app > app.py > get_data

```python

@app.route("/", methods=["POST"])
def add_data():
    """
    Adds new data
    """
    try:
        # Obtener los datos del cuerpo de la solicitud
        data = request.json

        # Validar que los datos no estén vacíos
        if not data:
            return jsonify({"err": "No data provided"}), 400

        # Validar campos requeridos (por ejemplo, "username")
        required_fields = ["username"]
        for field in required_fields:
            if field not in data:
                return jsonify({"err": f"Missing required field: {field}"}), 400

        # Crear una conexión
        conexion = mysql.connection.cursor()

        # Ejecutar la consulta de inserción
        query = "INSERT INTO user (username) VALUES (%s)"
        conexion.execute(query, (data["username"],))
        mysql.connection.commit()

        # Obtener el ID del nuevo registro
        new_id = conexion.lastrowid
        conexion.close()

        # Retornar respuesta con el ID creado
        return jsonify({"message": "User added successfully", "id_user": new_id}), 201

    except Exception as e:
        return jsonify({"err": f"Unexpected error: {str(e)}"}), 500


@app.route("/<int:id>", methods=["PATCH", "PUT"])
def modify_data(id):
    """
    Modifies existing data
    """
    return "Hola"


@app.route("/<int:id>", methods=["DELETE"])
def delete_data(id):
    """
    Deletes an item from the database
    """
    return "Hola"

```

1. Configuración de la base de datos

- Se configuran las credenciales de MySQL (MYSQL_HOST, MYSQL_USER, MYSQL_PASSWORD, MYSQL_DB) que el servidor Flask utilizará para conectarse a la base de datos.
- Se utiliza la extensión flask_mysqldb para manejar la conexión con la base de datos MySQL.

2. Ruta principal ("/") con método GET

- **Función get_data()**:
  - Esta ruta devuelve todos los usuarios almacenados en la tabla user.
  - Se realiza una consulta SQL (SELECT * FROM user) y se recorren los resultados para formatearlos en una lista de diccionarios con id_user y username.
  - La respuesta es devuelta como un objeto JSON.

3. Ruta con parámetro de ID ("/<int:id>") con método GET

- **Función get_one_by_id(id)**:
  - Esta ruta devuelve un usuario específico basado en su ID.
  - Realiza una consulta SQL para seleccionar el usuario cuyo id_user coincida con el valor proporcionado en la URL.
  - Si no se encuentra el usuario, devuelve un error 404 con un mensaje JSON. Si se encuentra, devuelve los datos del usuario en formato JSON.

4. Ruta principal ("/") con método POST

- **Función add_data()**:
  - Esta ruta permite agregar un nuevo usuario a la base de datos.
  - Los datos (en formato JSON) son extraídos del cuerpo de la solicitud.
  - Valida que los datos no estén vacíos y que el campo username esté presente.
  - Realiza una consulta SQL de inserción para agregar un nuevo usuario y devuelve un mensaje con el ID del nuevo usuario.

5. Ruta con parámetro de ID ("/<int:id>") con métodos PATCH y PUT

- **Función modify_data(id)**:
    - Esta ruta está diseñada para modificar los datos de un usuario existente. Actualmente, la función solo devuelve un mensaje de saludo ("Hola") sin implementar la lógica de actualización.

6. Ruta con parámetro de ID ("/<int:id>") con método DELETE

- **Función delete_data(id)**:
    - Esta ruta está diseñada para eliminar un usuario de la base de datos. Actualmente, la función solo devuelve un mensaje de saludo ("Hola") sin implementar la lógica de eliminación.

Dockerfile

app > Dockerfile > ...

```
1   # Usa una imagen base de Python
2   FROM python:3.10
3
4   # Define el directorio de trabajo
5   WORKDIR /app
6
7   # Copia los archivos de la app al contenedor
8   COPY . /app
9
10  # Instala las dependencias
11  RUN pip install -r requirements.txt
12
13  # Expone el puerto
14  EXPOSE 5000
15
16  # Define el comando para correr la app
17  CMD ["python", "app.py"]
18
```

- **FROM python:3.10**: Usa una imagen base de Python 3.10.
- **WORKDIR /app**: Define el directorio de trabajo dentro del contenedor como /app.
- **COPY . /app**: Copia todos los archivos del directorio actual al directorio /app dentro del contenedor.
- **RUN pip install -r requirements.txt**: Instala las dependencias de la aplicación desde el archivo requirements.txt.
- **EXPOSE 5000**: Expone el puerto 5000, que es donde la aplicación escuchará las peticiones.
- **CMD ["python", "app.py"]**: Define el comando para ejecutar la aplicación, iniciando app.py con Python.

Requirements.txt

app > requirements.txt

```
1  flask
2
3  flask-mysqldb
```

Flask es un microframework para Python que permite desarrollar aplicaciones web de manera sencilla y flexible. Es popular por su simplicidad y facilidad de uso, especialmente en aplicaciones pequeñas y medianas. Flask permite agregar funcionalidades a través de extensiones, como la base de datos, autenticación, formularios, etc.

Flask-MySQLdb es una extensión para Flask que permite integrar fácilmente MySQL con una aplicación Flask. Utiliza el conector MySQLdb de Python para facilitar la conexión y la ejecución de consultas SQL dentro de una aplicación Flask.

Docker-compose.yaml

docker-compose.yaml

```yaml
services:
  database:
    image: mysql:8.0
    ports:
      - 3307:3306
    volumes:
      - ./mysql:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: C413b  # Cambia esta contraseña por una segura
      MYSQL_DATABASE: prueba    # Nombre de la base de datos inicial
      MYSQL_USER: vane        # Usuario inicial de MySQL
      MYSQL_PASSWORD: C413b


  flask:
    build: ./app
    ports:
      - 5000:5000
    depends_on:
      - database
```

Servicio database:

image: mysql:8.0: Utiliza la imagen oficial de Docker para MySQL en su versión 8.0.

ports: 3307:3306: Mapea el puerto 3307 del host (máquina local) al puerto 3306 del contenedor, que es el puerto por defecto de MySQL.

volumes: ./mysql:/var/lib/mysql: Monta el directorio local ./mysql en el directorio /var/lib/mysql dentro del contenedor, asegurando que los datos de la base de datos se guarden de manera persistente en tu máquina local.

environment: Define las variables de entorno para la configuración de MySQL:

MYSQL_ROOT_PASSWORD: Establece la contraseña del usuario root de MySQL (debes reemplazar esto con una contraseña segura).

MYSQL_DATABASE: Crea una base de datos inicial llamada prueba.

MYSQL_USER: Define un usuario inicial de MySQL llamado vane.

MYSQL_PASSWORD: Establece la contraseña del usuario vane (debes definirla aquí).

2. Servicio flask:

build: ./app: Docker construye la imagen del contenedor Flask desde el directorio local ./app, donde se espera que estén los archivos de la aplicación Flask (como el código fuente y un Dockerfile).

ports: 5000:5000: Mapea el puerto 5000 del contenedor Flask al puerto 5000 de la máquina local. Esto significa que la aplicación Flask será accesible desde http://localhost:5000 en el navegador.

depends_on: - database: Indica que el servicio flask depende del servicio database. Esto asegura que el contenedor de la base de datos se inicie antes que el contenedor de la aplicación Flask.

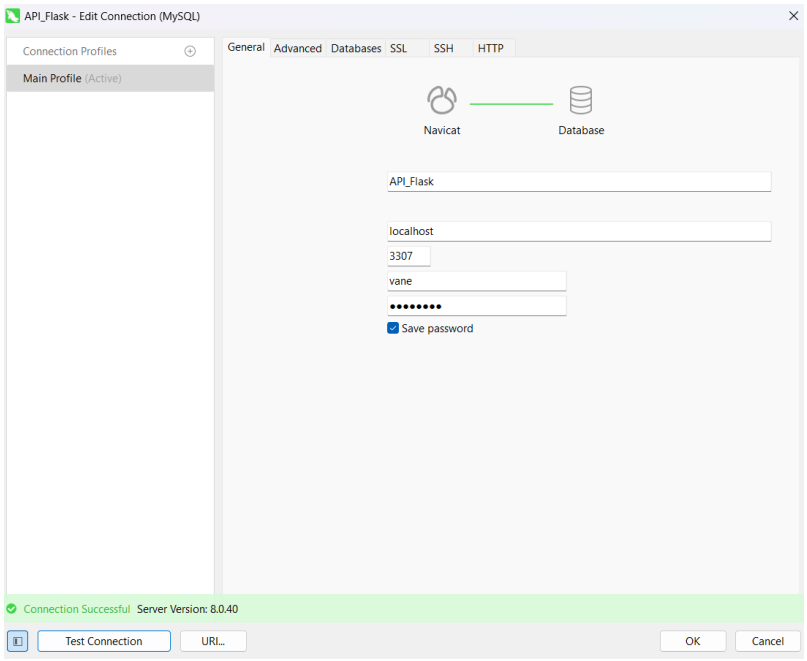
NAVICAT

1. Corremos el programa con la siguiente expresión:

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\vanes\OneDrive\Escritorio\plantilla-flask> docker compose up --build -d
[+] Building 2.7s (9/9) FINISHED
 => [flask internal] load build definition from Dockerfile
 => => transferring dockerfile: 372B
 => [flask internal] load metadata for docker.io/library/python:3.10
 => [flask internal] load .dockerignore
 => => transferring context: 2B
 => [flask 1/4] FROM docker.io/library/python:3.10@sha256:3ba2e48b887586835af6a0c35fc6fc6086fb4881e963082330ab0a35f3f42c16
 => => resolve docker.io/library/python:3.10@sha256:3ba2e48b887586835af6a0c35fc6fc6086fb4881e963082330ab0a35f3f42c16
 => [flask internal] load build context
 => => transferring context: 93B
 => CACHED [flask 2/4] WORKDIR /app
 => CACHED [flask 3/4] COPY . /app
 => CACHED [flask 4/4] RUN pip install -r requirements.txt
 => [flask] exporting to image
 => => exporting layers
 => => writing image sha256:a52c724a6e88ea6e6724b7eb3defc13f6f11751111b530801f3a6711521ad81c
 => => naming to docker.io/library/plantilla-flask-flask
[+] Running 3/3
 ✓ Network plantilla-flask_default       Created
 ✓ Container plantilla-flask-database-1  Started
 ✓ Container plantilla-flask-flask-1     Started
PS C:\Users\vanes\OneDrive\Escritorio\plantilla-flask>
```

2.- Creamos una nueva conexión en Navicat



3. Dentro de Navicat creamos una query para crear una tabla usuario con los siguientes atributos:

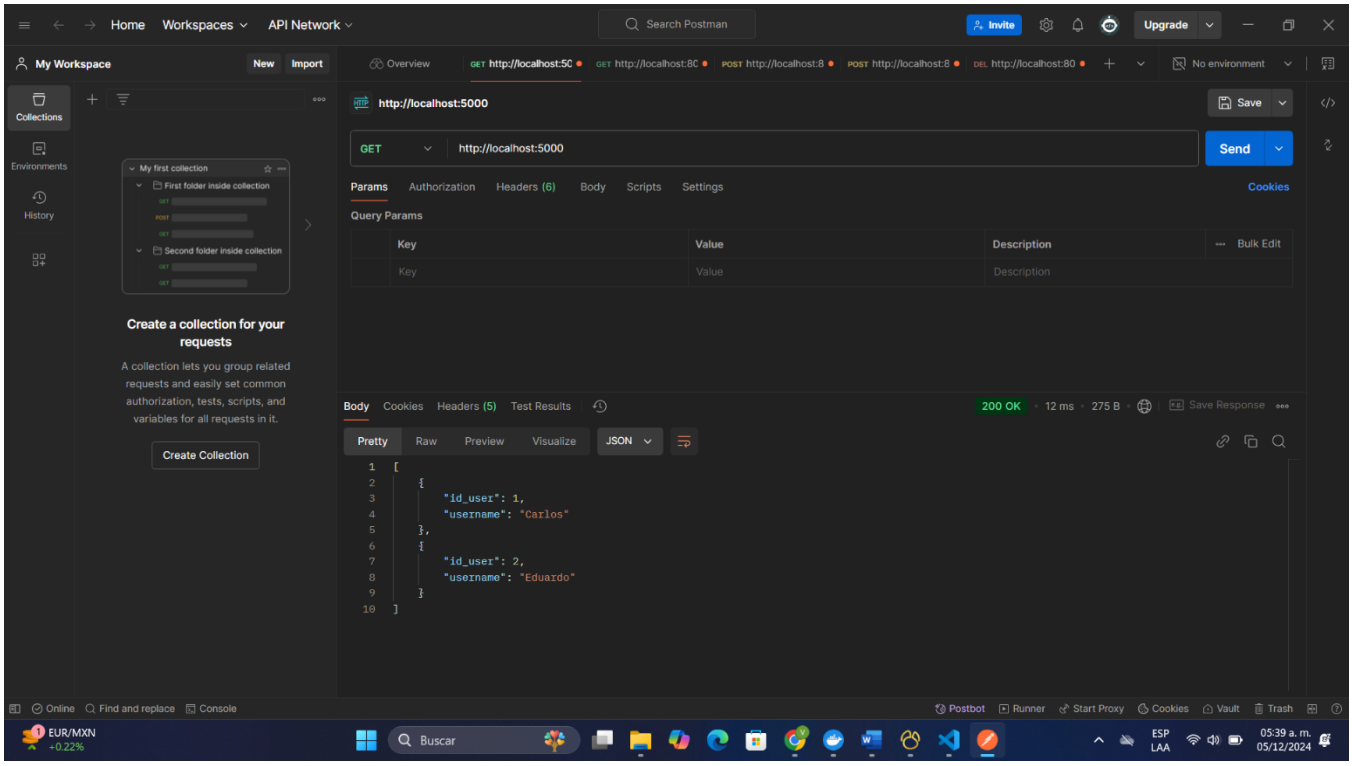
```
CREATE TABLE user(
    id_user INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(100)
);
```

4. Ingresamos dos usuarios

```
INSERT INTO user (username) VALUES ('Carlos'), ('Eduardo');
```

POSTMAN

5. Colocamos la siguiente dirección para visualizar los usuarios



Conclusión

Este conjunto de códigos configura una solución completa para una aplicación web utilizando Flask y MySQL, gestionada con Docker. La API en Flask permite interactuar con una base de datos MySQL, realizando operaciones básicas como obtener, insertar y modificar datos. Docker se utiliza para contenerizar ambos servicios, asegurando que la base de datos y la aplicación web se ejecuten de manera aislada pero conectada.

El archivo Dockerfile define cómo construir la imagen para la aplicación Flask, configurando Python y las dependencias necesarias. Por otro lado, docker-compose.yml gestiona la configuración de los contenedores, incluyendo la base de datos MySQL con su configuración inicial (usuario, contraseña, base de datos) y el contenedor de Flask que depende del primero.