MELP

MANUAL TÉCNICO

Carlos Augusto Calderón Estrada

11/11/2021

Objetivos

General:

Brindarle apoyo al desarrollador a trabajar en esta API como una guía lo más completa y sencilla posible para el mejor entendimiento de lo que se lleva desarrollado en la API y así evitar errores posibles por el uso incorrecto de los algoritmos y/o sentencias.

Específicos:

- Que el desarrollador trabaje en la API como si fuera creada por el/ella.
- Proporcionarle al lector una explicación sencilla y técnica de los procesos algorítmicos y las relaciones de los métodos, funciones y atributos que son esenciales en la API.

Introducción

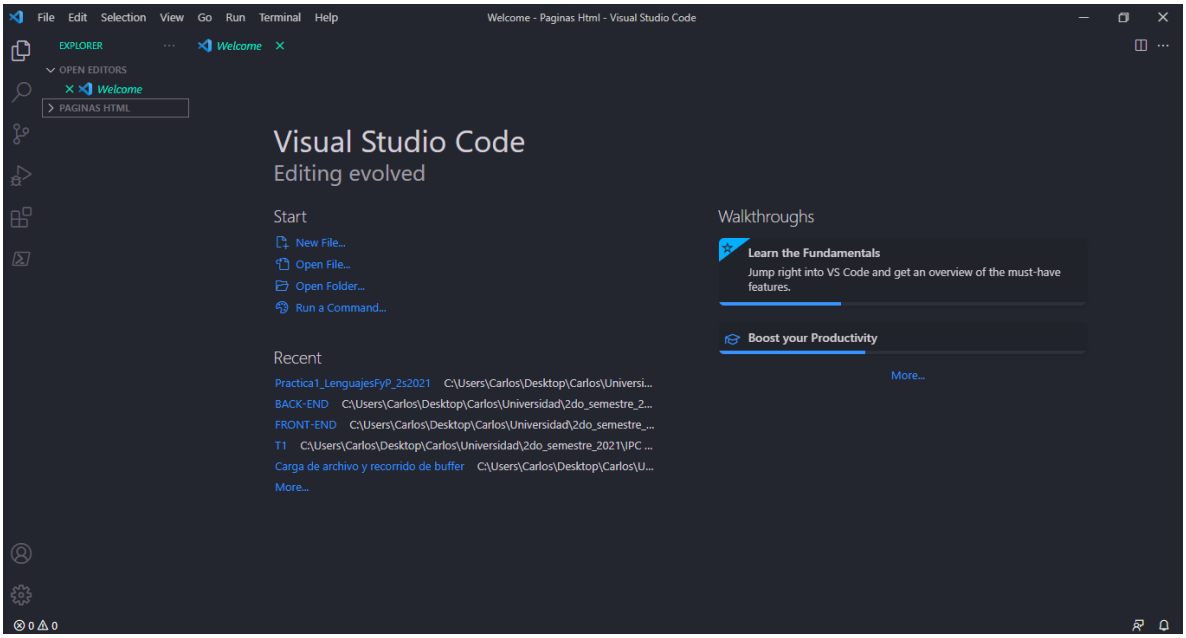
Este manual técnico tiene como fin dar a conocer al desarrollador las mejores recomendaciones, requerimientos para un mejor entendimiento al momento de querer realizar modificaciones, optimización de procesos o métodos, indicando el editor de texto que se utilizó para el desarrollo de dicha aplicación.

La API se basa en conectar con una base de datos, se contienen distintas rutas representando los procesos **CRUD** (create, read, update, delete), y una ruta para calcular estadísticas recibiendo como parámetros unas coordenadas y un radio, esta API fue desarrollada por medio de lenguaje de programación **Python**, utilizando el **framework Flask.**

Para la prueba de la API se ha exportado una colección de **postman** y almacenado en la carpeta misma donde se ubica este documento, podrá importarla y hacer un test de cada ruta creada.

Editor de texto utilizado

1. El editor de texto utilizado fue Visual Studio Code gracias a la variedad de extensiones que posee y su fácil uso para el manejo de la nube git, que permite agregar la propia terminal y ejecutar los comandos desde el entorno de VS Code.



Requerimientos:

- Procesador de 1.6 GHz o más.
- 1 GB de memoria RAM
- Plataformas aprobadas: OS X, Windows 7, Linux (Debian), Linux (Red Hat)
- Microsoft .NET Framework 4.5.2 para VS Code

Sistema Operativo que se llevó a cabo:

Windows 10 de 64 bits

Lógica del Programa

Main:

Clase que almacena las rutas e inicializa el servidor local

```
app = Flask(__name__)
app.config["DEBUG"] = True
#Iniciamos el servidor
if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True )
```

Init Gestor:

Se inicia la conexión y host: Aquí se debe configurar el usuario, la contraseña y el nombre de la base de datos a utilizar.

```
def __init__(self):#connection of the python with mysql database
        self.connection = mysql.connector.connect(user='root', password='',
host='localhost', database='melp', port='3306')
        self.midb = self.connection.cursor()
```

RUTAS

ViewData:

Inicia una función en la clase Gestor() que retorna los registros de la tabla mysql en formato json.

```
@app.route('/ViewData', methods=['GET'])#view data method READ
def getData():
    return gestor.ViewRegisters()
```

función ViewRegisters de la clase Gestor():

En este método se realiza el algoritmo para seleccionar, obtener y retornar los registros de la base de datos de la tabla restaurants

```
    def ViewRegisters(self):
        view = 'SELECT * FROM restaurants'
        self.midb.execute(view)
        field_names = [i[0] for i in self.midb.description]#campos
        restaurants = self.midb.fetchall()
        json_data=[]
        for i in restaurants:
```

```
            json_data.append(dict(zip(field_names, i)))
        return json.dumps(json_data)
```

CreateRegister:

Inicia una función que recibe los datos en formato json y los envía a la clase Gestor() que crea un nuevo registro en la tabla mysql.

```
@app.route('/CreateRegister', methods=['POST'])#method CREATE
def insertRegister():
    data = request.json
    gestor.InsertRegister(data['id'], data['rating'], data['name'],
data['site'], data['email'], data['phone'], data['street'], data['city'],
data['state'], data['lat'], data['lng'])
    return '{"Estado":"Registro creado"}'
```

función InsertRegister de la clase Gestor():

Recibe todos los parámetros de la tabla de la base de datos por medio de la ruta (función anterior) y genera una consulta en la base de datos creando el nuevo registro y guardándolo(commit).

```
def InsertRegister(self, iid, rating, name, site, email, phone, street,
city, state, lat, lng ):
        insert = "INSERT INTO restaurants(id, rating, name, site, email,
phone, street, city, state, lat, lng) VALUES
('{}','{}','{}','{}','{}','{}','{}','{}','{}','{}','{}')".format(iid,
rating, name, site, email, phone, street, city, state, lat, lng)
        self.midb.execute(insert)
        self.connection.commit()#save the changes in the database
```

UpdateRegister:

Inicia una función que recibe un parámetro **id** por medio de la ruta, esta función a su vez recibe los datos que se actualizarán en formato json y los envía a la clase Gestor() que actualiza un registro en la tabla mysql. En este caso solo actualiza los campos que un restaurante puede cambiar (name, email, phone)

```
@app.route('/UpdateRegister/<id>', methods=['PUT'])#method UPDATE
def UpdateRegister(id):
    data = request.json
    print(data)
    gestor.updateRegister(id,data['name'],data['email'], data['phone'])
    return '{"Estado":"Registro actualizado"}'
```

función UpdateRegister de la clase Gestor():

Recibe el parámetro **id, nameNuevo, emailNuevo, phoneNuevo,** que buscara en la tabla de la base de datos por medio del **id** en la consulta indicada (ver código siguiente) y esta misma actualizará el registro que coincida con el **id** finalizando y guardando la base de datos (commit).

```
def updateRegister(self, id,nameN, emailN, phoneN):# update register in the
data base in the fields name, email, phone
        update1 = "UPDATE restaurants SET name='{}' WHERE
id='{}'".format(nameN, id)
        update2 = "UPDATE restaurants SET email='{}' WHERE
id='{}'".format(emailN, id)
        update3 = "UPDATE restaurants SET phone='{}' WHERE
id='{}'".format(phoneN, id)
        self.midb.execute(update1)
        self.midb.execute(update2)
        self.midb.execute(update3)
        self.connection.commit()
```

Delete:

Inicia una función que recibe los datos en formato json, específicamente el dato **id** y lo envía a la clase Gestor() que remueve un nuevo registro en la tabla mysql.

```
@app.route('/Delete', methods=['DELETE'])#method DELETE
def DeleteRegister():
    data = request.json
    gestor.DeleteRegister(data['id'])
    return '{"Estado":"Registro Eliminado"}'
```

función DeleteRegister de la clase Gestor():

Recibe el parámetro id que buscara en la tabla de la base de datos por medio de la consulta indicada (ver código siguiente) y esta misma eliminara el registro que coincida con el **id** finalizando y guardando la base de datos (commit).

```
def DeleteRegister(self, id):# function that delete register in the data
base
        delete = "DELETE FROM restaurants WHERE id='{}'".format(id)
        self.midb.execute(delete)
        self.connection.commit()
```

Alredy/statistics:

Inicia una función que recibe los datos como parametros, específicamente el dato **latitude, longitude y radius**, estos los envía a la clase Gestor() que se encarga de realizar los procesos estadísticos.

```
@app.route('/Alredy/statistics', methods=['GET'])
def Alredy():
    latitude = request.args.get('latitude')
    longitude = request.args.get('longitude')
    radius = request.args.get('radius')
    resultado=gestor.Alredy(latitude,longitude,radius)
    return resultado
```

función Alredy de la clase Gestor():

Inicia una función que recibe los parámetros a utilizar, específicamente el dato **latitude, longitude y radius**, luego inicializa una lista que almacenara los **rating**, esta función contiene las consultas para obtener todas las latitudes y longitudes de la base de datos, en el que por medio de un while se recorre dichas tuplas, enviando datos a una función que retorna la distancia entre la coordenada inicial y la coordenada a calcular, luego se compara el resultado con la distancia y se determina si el restaurante pertenece dentro del círculo, almacenándolos dentro de la lista rating (si es que se determina que si pertenece) para luego enviar a otra función que obtiene los datos de todos los ratings que coincidan con cada coordenada para poder iniciar el conteo, el promedio y la desviación estándar.

```
def Alredy(self, lat1, lng1, radius):
        listaR=[]
        lati = "SELECT lat FROM restaurants"#consulta
        self.midb.execute(lati)
        lats = self.midb.fetchall()
        longi = "SELECT lng FROM restaurants"#consulta
        self.midb.execute(longi)
        longs = self.midb.fetchall()
        i=0
        while(i<len(lats)):#ciclo para recorrer todos los registros
            lat2=lats[i]
            lat2 = lat2[0]
            lng2=longs[i]
            lng2= lng2[0]
            distance = round(self.Calculos(float(lat1), float(lng1),
float(lat2),float(lng2)),2)
            if distance<=int(radius):#determinamos si pertenece al circulo
                rating = self.GetRating(lat2,lng2)
                if rating is not None:#agregamos a la lista rating
                    listaR.append(rating)
            i+=1
        if len(listaR)!=0:
            count = len(listaR)
            ratingProm = round((sum(listaR)/count),2
            ratingdesv = round(statistics.pstdev(listaR),2)
            resultado='{"Count":"'+str(count)+'",
"avg":"'+str(ratingProm)+'", "std":"'+str(ratingdesv)+'"}'
            return resultado
        return '{"No exist restaurants alrady, try with other radius"}'
```

Librerías Utilizadas

Las librerías que se utilizaron para el desarrollo de esta practica fueron:

Imports locales:

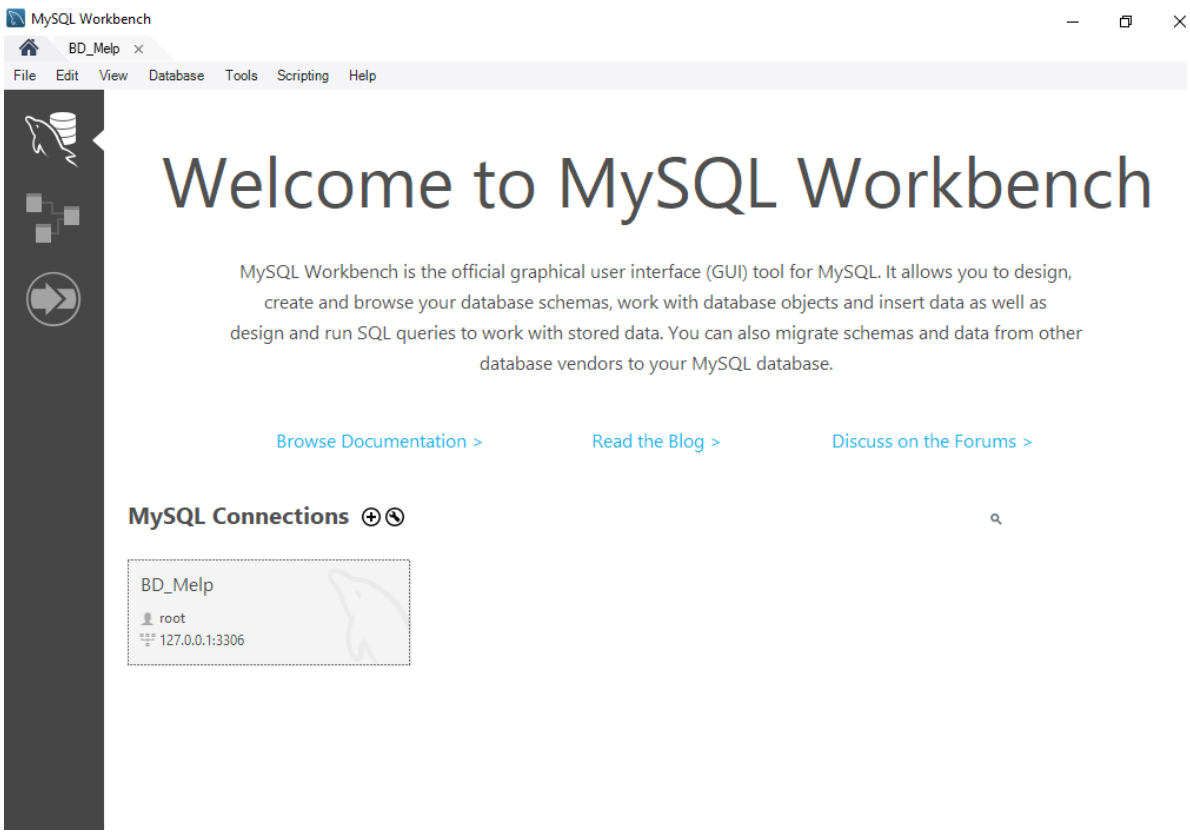
- Import mysql.connector
- Import json
- Import math, statistics
- From Gestor import Gestor

Imports Flask:

- From flask import *
- From flask_cors import CORS

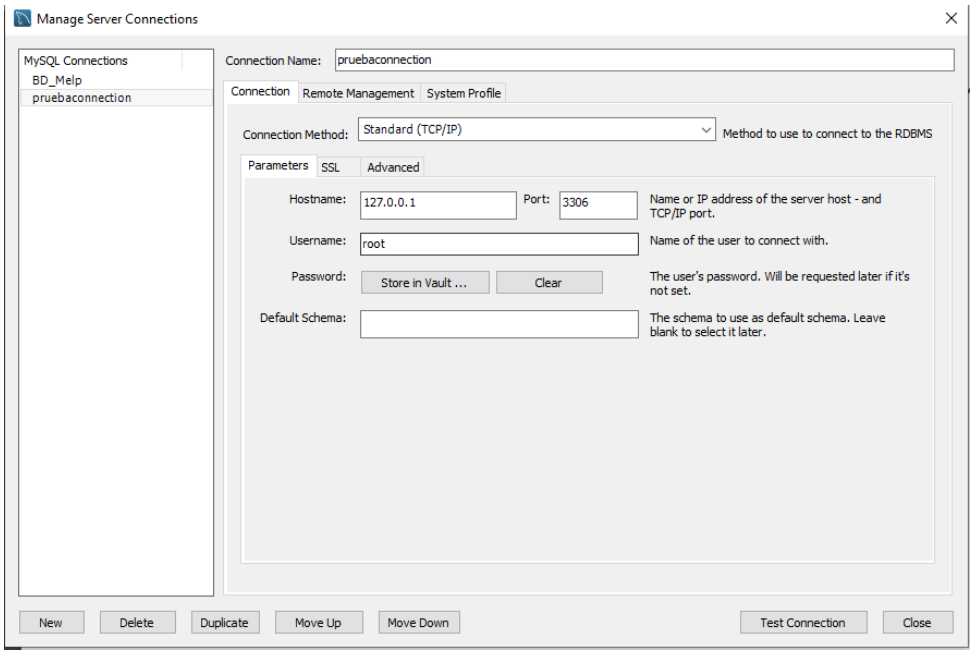
WorkBench

1. Abrimos workbench:

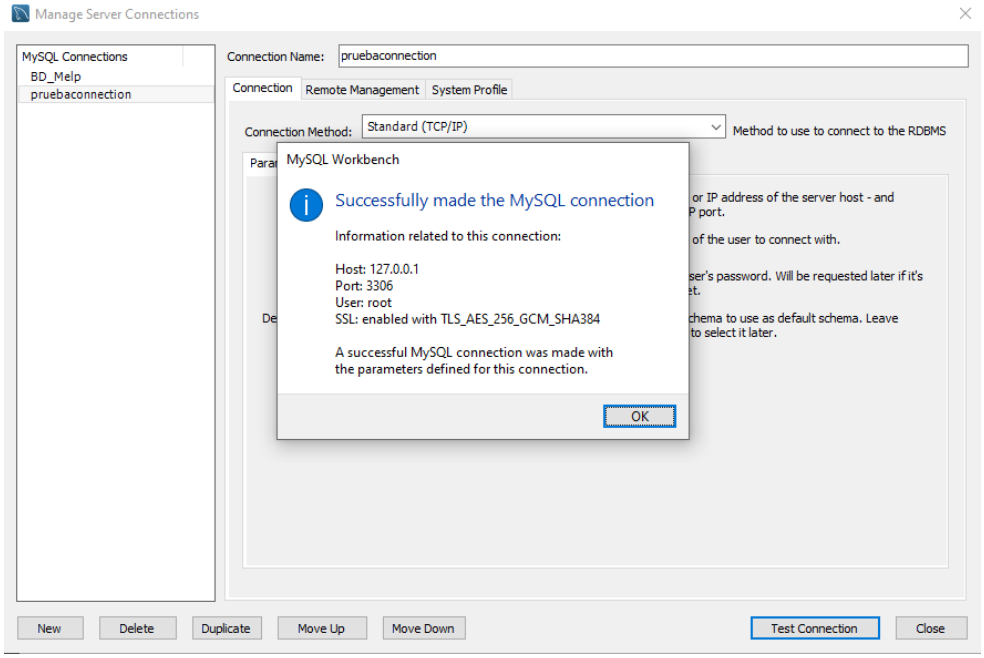


2. **Nos dirigimos en el boton +:**

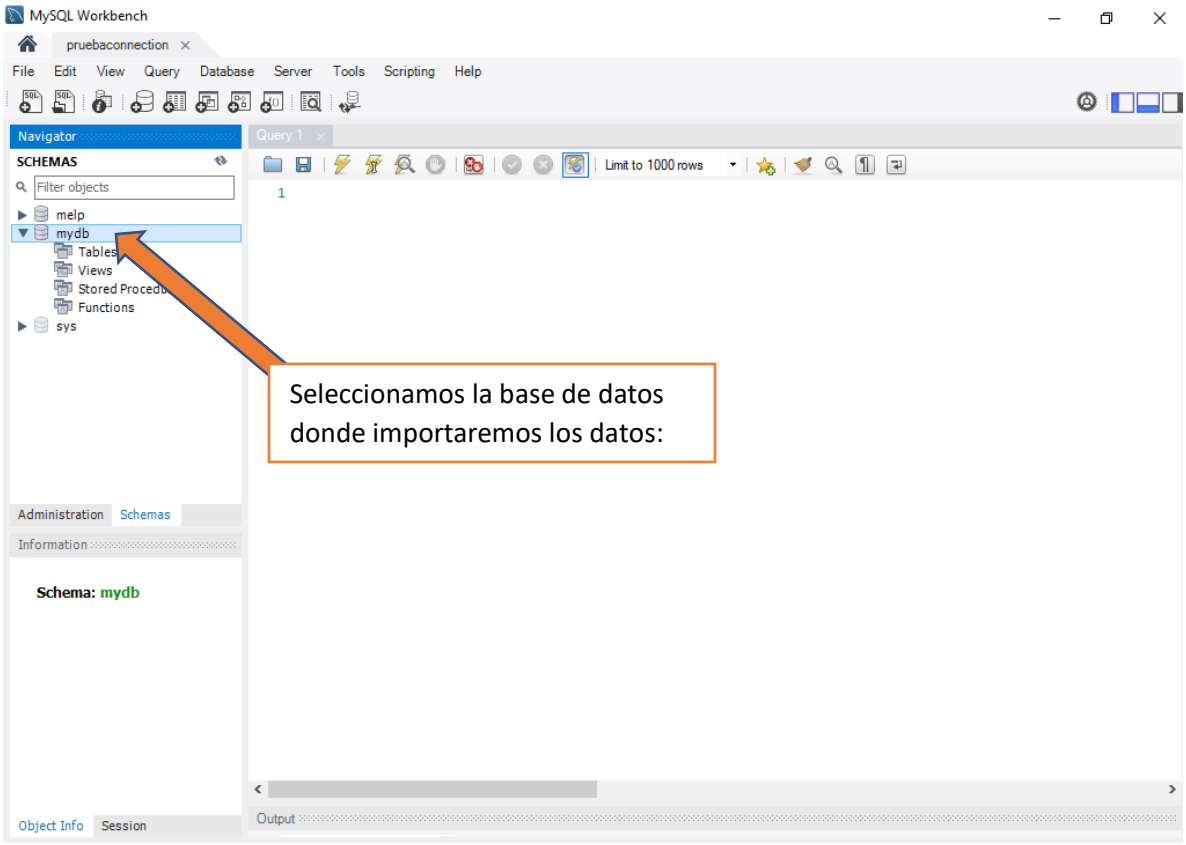
   Determinamos el nombre del server y la contraseña del usuario



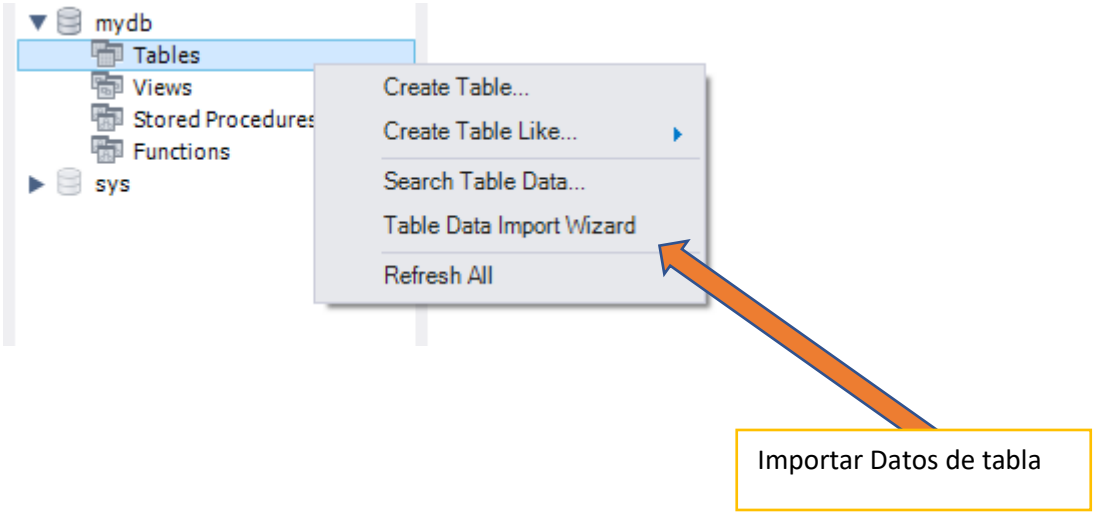
3. Se testea la conexión



4.

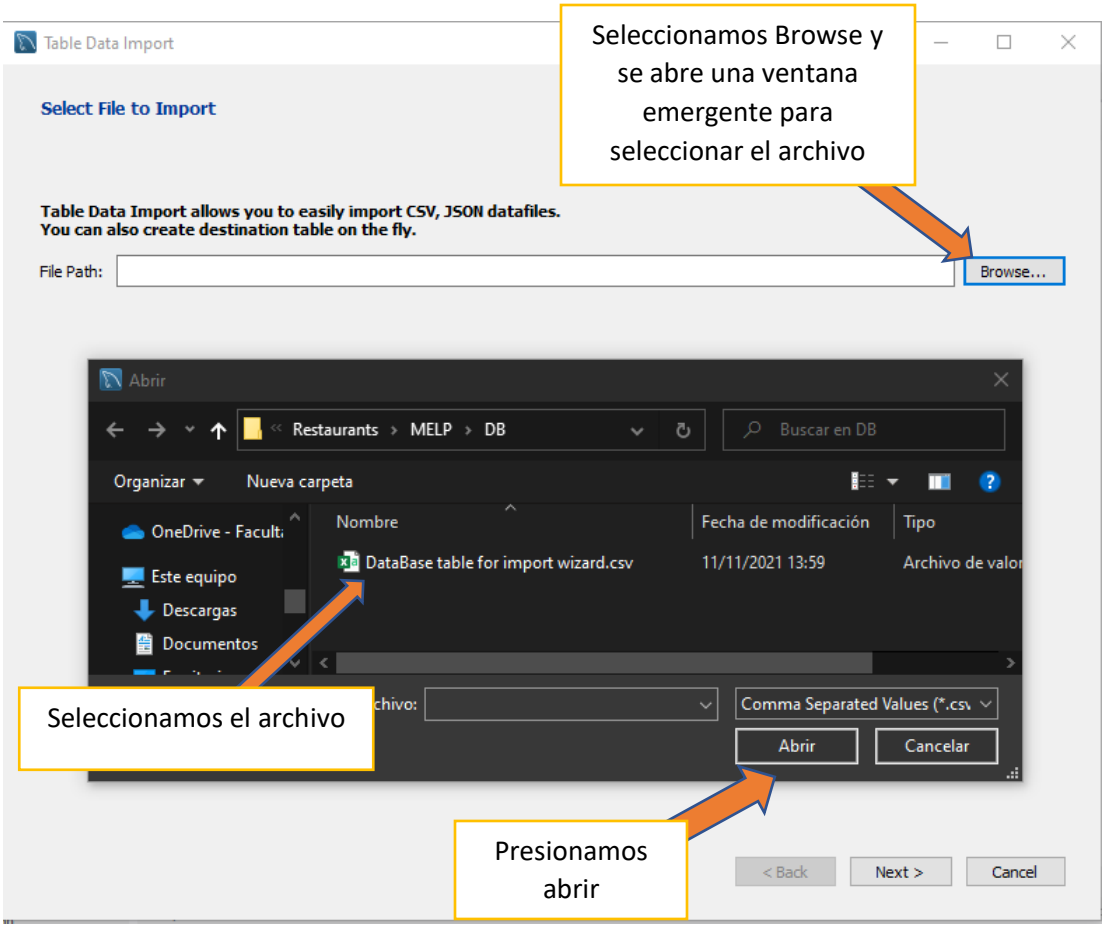


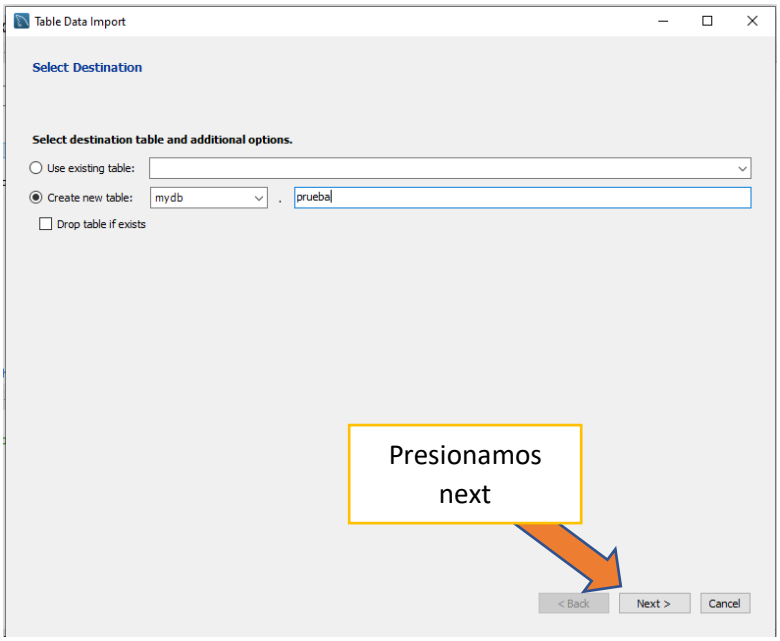
5. Hacemos click derecho en las tablas



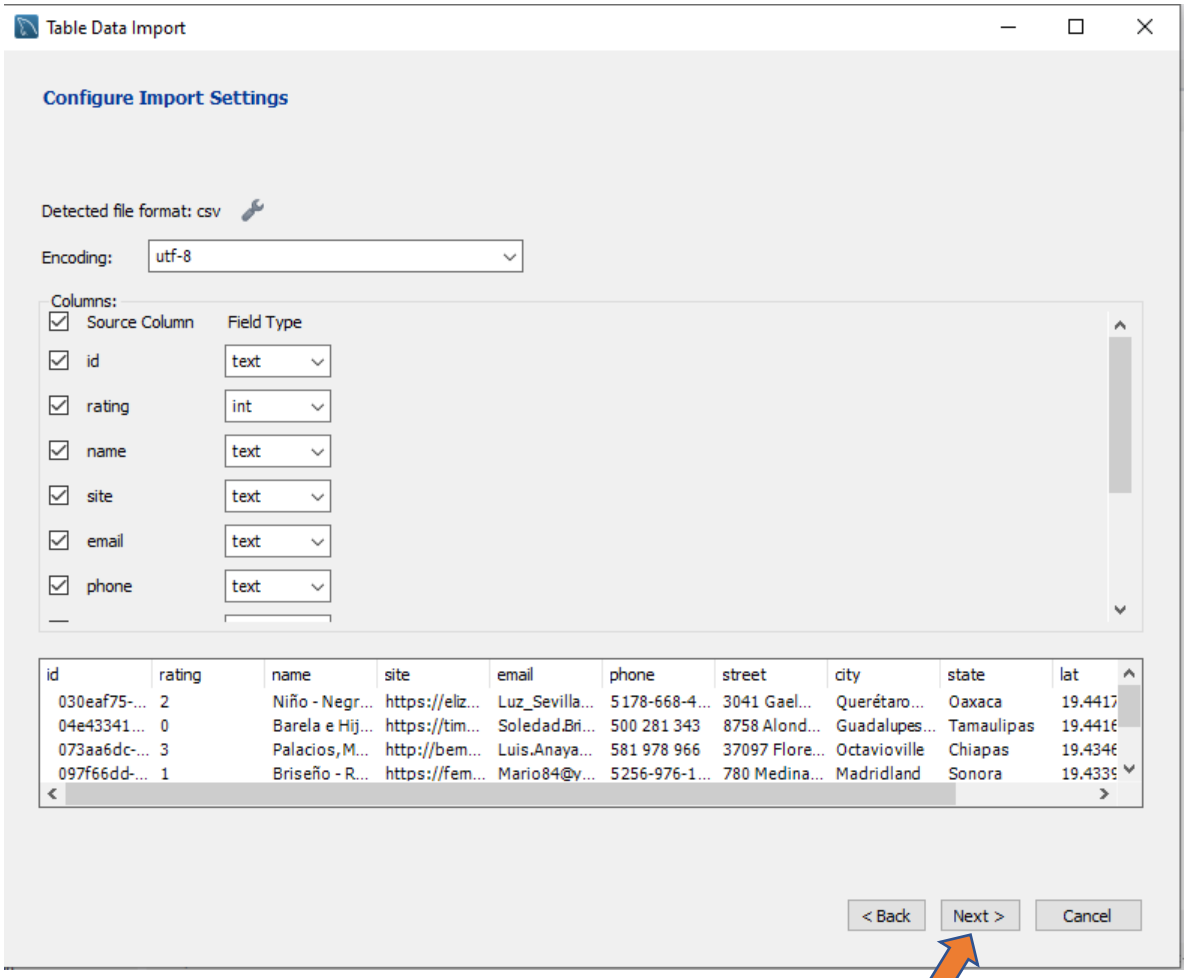
6.

7.

8. 
Table Data Import
Select File to Import
Table Data Import allows you to easily import CSV, JSON datafiles.
You can also create destination table on the fly.
File Path:
Browse...
Seleccionamos Browse y se abre una ventana emergente para seleccionar el archivo
Abrir
Restaurants › MELP › DB
Buscar en DB
Organizar
Nueva carpeta
OneDrive - Faculti
Este equipo
Descargas
Documentos
Nombre
Fecha de modificación
Tipo
DataBase table for import wizard.csv
11/11/2021 13:59
Archivo de valor
Seleccionamos el archivo
Comma Separated Values (*.csv
Abrir
Cancelar
Presionamos abrir
< Back
Next >
Cancel

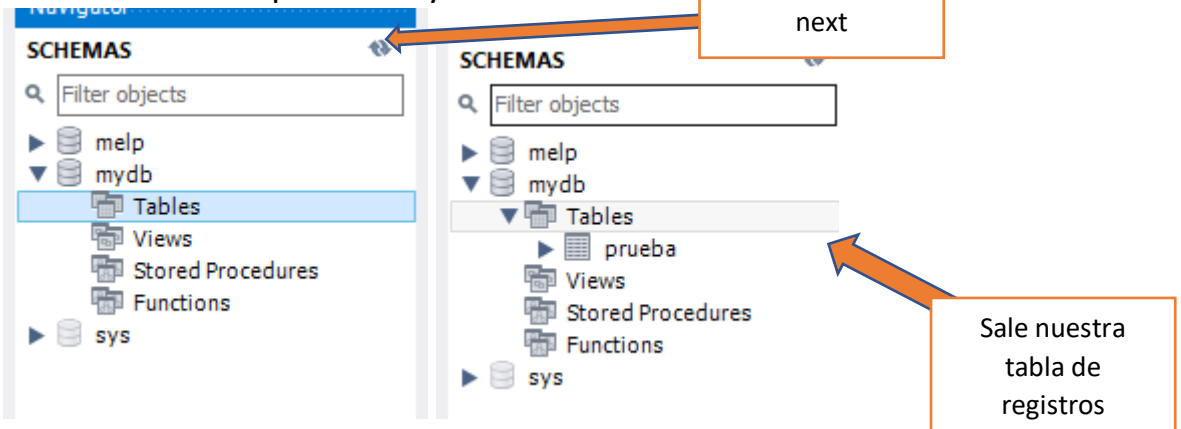
9. 
Table Data Import
Select Destination
Select destination table and additional options.
Use existing table:
Create new table:
mydb
prueba
Drop table if exists
Presionamos next
< Back
Next >
Cancel

10.



Presionamos next

11. Finalizamos la importación y actualizamos



Presionamos next

Sale nuestra tabla de registros