MELP

MANUAL TÉCNICO

Carlos Augusto Calderón Estrada

11/11/2021

Objetivos

General:

Provide support to the developer to work on this API as a guide as complete and simple as possible for a better understanding of what has been developed in the API and thus avoid possible errors due to the incorrect use of algorithms and / or sentences.

Specific:

- That the developer works in the API as if it were created by him / her.
- Provide the reader with a simple and technical explanation of the algorithmic processes and the relationships of the methods, functions and attributes that are essential in the API.

Introduction

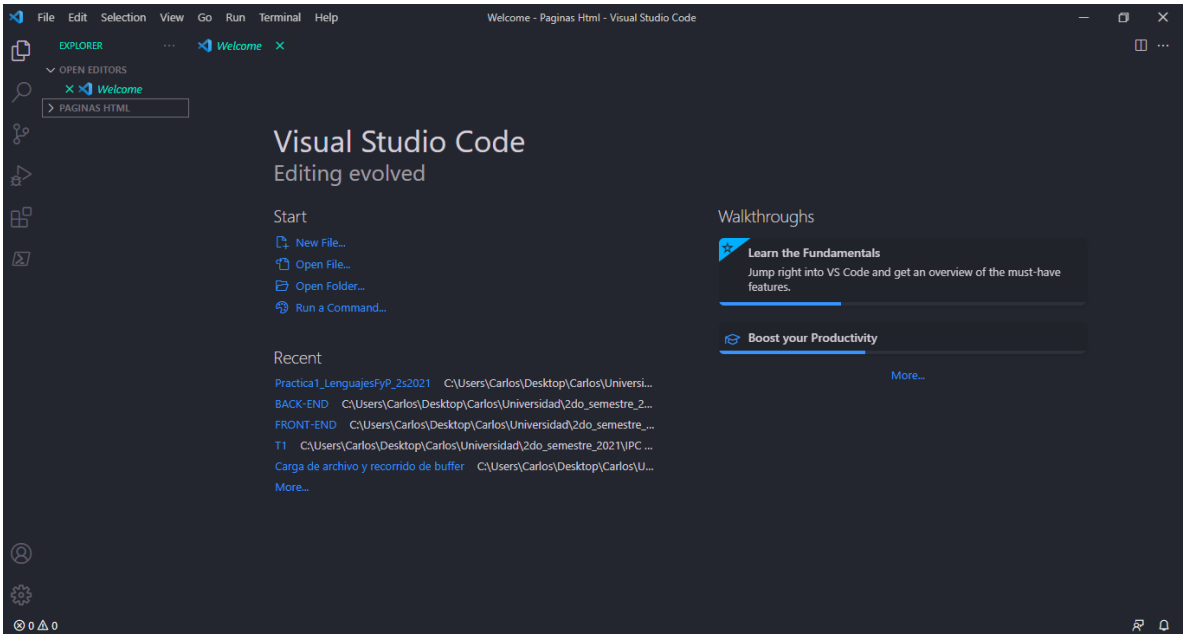
The purpose of this technical manual is to inform the developer of the best recommendations, requirements for a better understanding when wanting to make modifications, optimization of processes or methods, indicating the text editor that was used for the development of said application.

The API is based on connecting to a database, different routes are contained representing the processes **CRUD** (create, read, update, delete), and a route to calculate statistics receiving as parameters some coordinates and a radius, this API was developed through the programming language **Python**, using the **framework Flask.**

For the API test a collection of **postman** and stored in the same folder where this document is located, you can import it and make a test of each route created.

Editor de texto utilizado

1. The text editor used was Visual Studio Code thanks to the variety of extensions it has and its easy use for managing the git cloud, which allows adding the terminal itself and executing the commands from the VS Code environment.



Requerirements:

- Processor1.6 GHz or more.
- 1 GB of memory RAM
- Approved platforms: OS X, Windows 7, Linux (Debian), Linux (Red Hat)
- Microsoft .NET Framework 4.5.2 para VS Code

Operating System that was carried out:

Windows 10 de 64 bits

Program logic

Main:

Class that stores the routes and initializes the local server

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

Init Gestor:

The connection and host starts: Here you must configure the user, password and name of the database to use.

```
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:

Start a function in the Manager () class that returns the records from the mysql table in format json.

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

función ViewRegisters de la clase Gestor():

In this method, the algorithm is carried out to select, obtain and return the records of the database from the restaurants table

```
    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:

Starts a function that receives the data in json format and sends it to the Manager () class that creates a new record in the mysql table.

```
@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():

It receives all the parameters of the database table through the path (previous function) and generates a query in the database creating the new record and saving it (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:

It starts a function that receives an id parameter through the path, this function in turn receives the data to be updated in json format and sends it to the Manager () class that updates a record in the mysql table. In this case it only updates the fields that a restaurant can change (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():

Receive the parameter **id, nameNuevo, emailNuevo, phoneNuevo,** It will search the database table by means of the id in the indicated query (see the following code) and it will update the record that matches the **id**, ending and saving the database (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:

It starts a function that receives the data in json format, specifically the **id** data and sends it to the Manager () class that removes a new record in the mysql table.

```
@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():

It receives the id parameter that it will search in the database table by means of the indicated query (see following code) and this same one will eliminate the record that matches the id ending and saving the database (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:

It starts a function that receives the data as parameters, specifically the data latitude, longitude and radius, these are sent to the Manager () class that is in charge of carrying out the statistical processes.

```
@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():

It starts a function that receives the parameters to be used, specifically the latitude, longitude and radius data, then initializes a list that will store the ratings, this function contains the queries to obtain all the latitudes and longitudes from the database, in which by Through a while, these tuples are traversed, sending data to a function that returns the distance between the initial coordinate and the coordinate to be calculated, then the result is compared with the distance and it is determined if the restaurant belongs within the circle, storing them within the rating list (if it is determined that it belongs) and then send it to another function that obtains the data of all the ratings that coincide with each coordinate in order to start the count, the average and the standard deviation.

```
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"}'
```

Libraries used

The libraries that were used for the development of this practice were:

Imports locales:

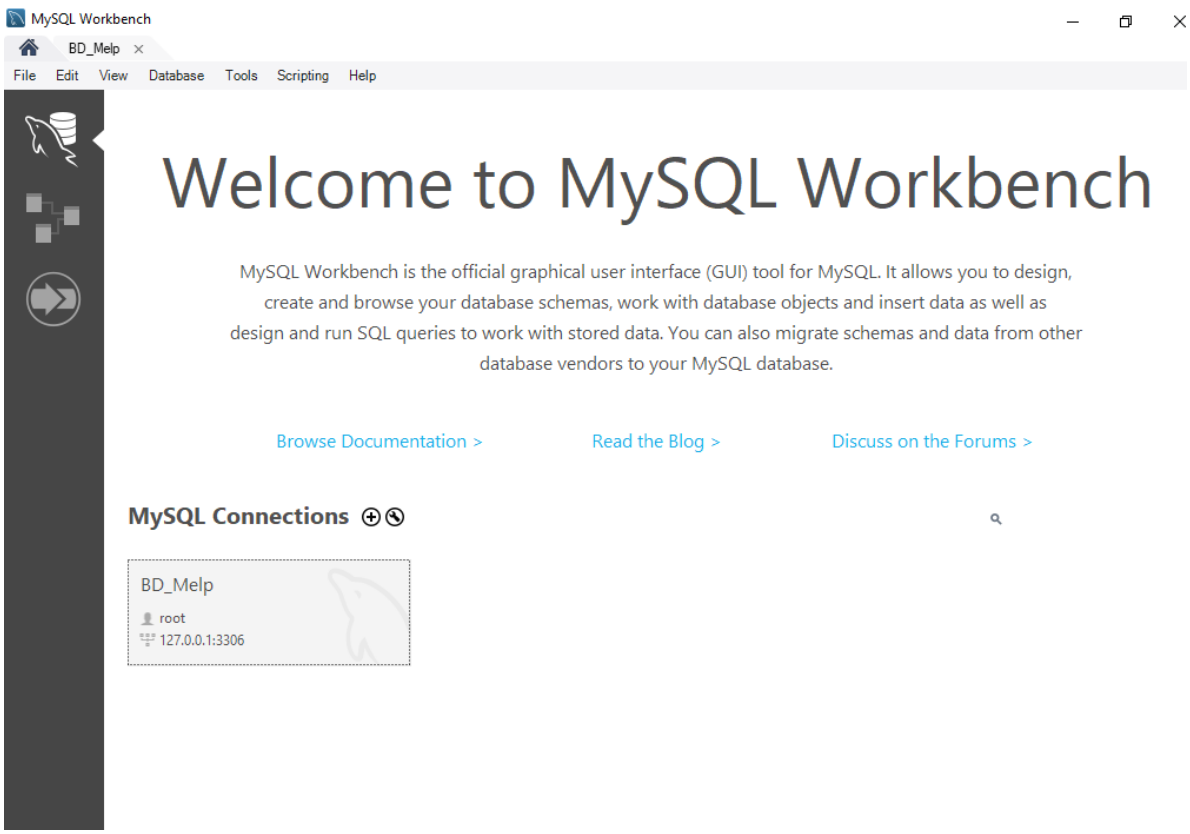
- Import mysql.connector
- Import json
- Import math, statistics
- From Gestor import Gestor

Imports Flask:

- From flask import *
- From flask_cors import CORS

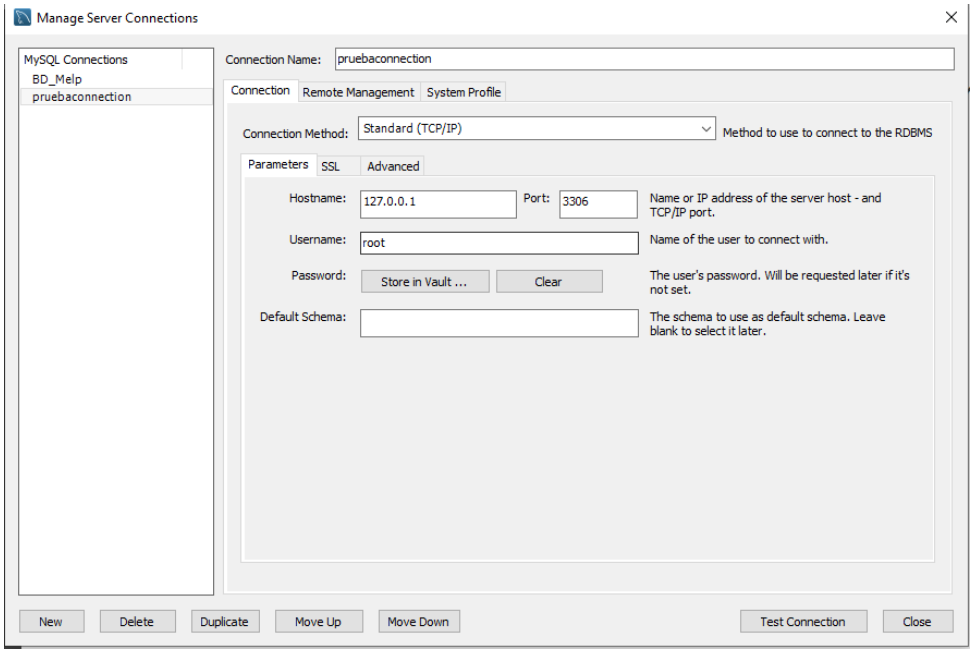
WorkBench

1. Abrimos workbench:

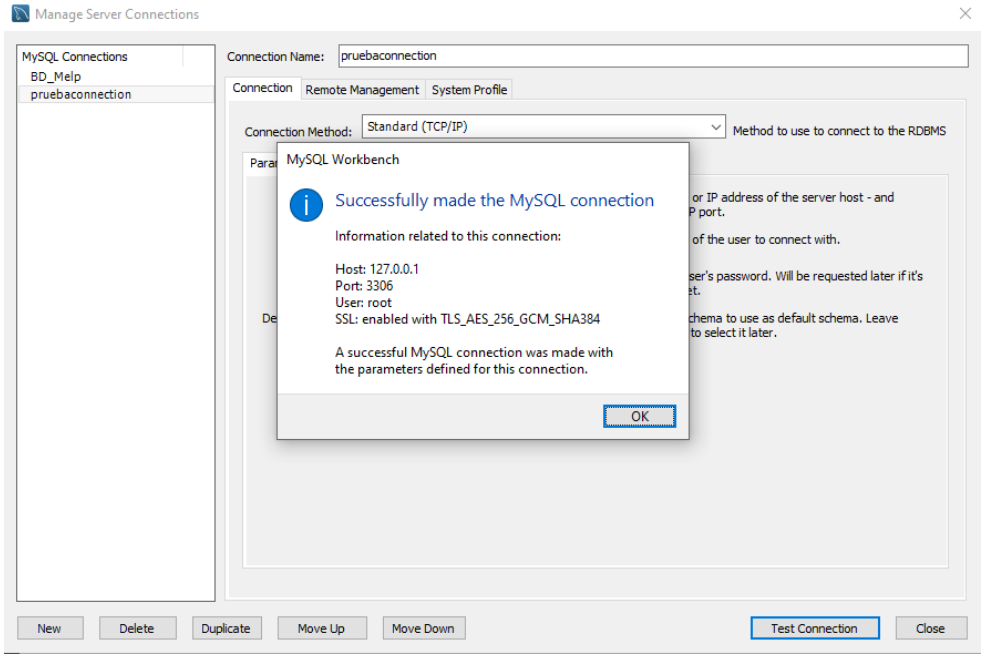


2. **We go to the + button:**

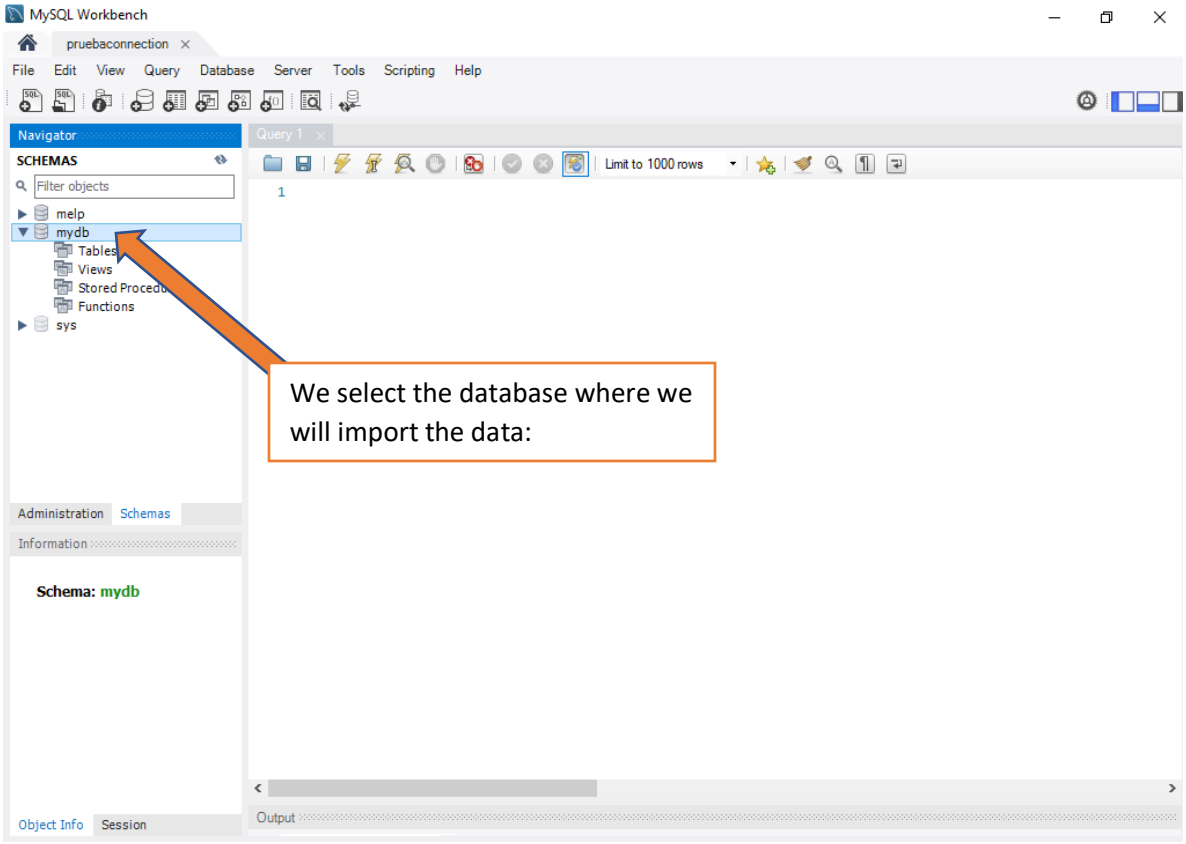
   We determine the name of the server and the password of the user



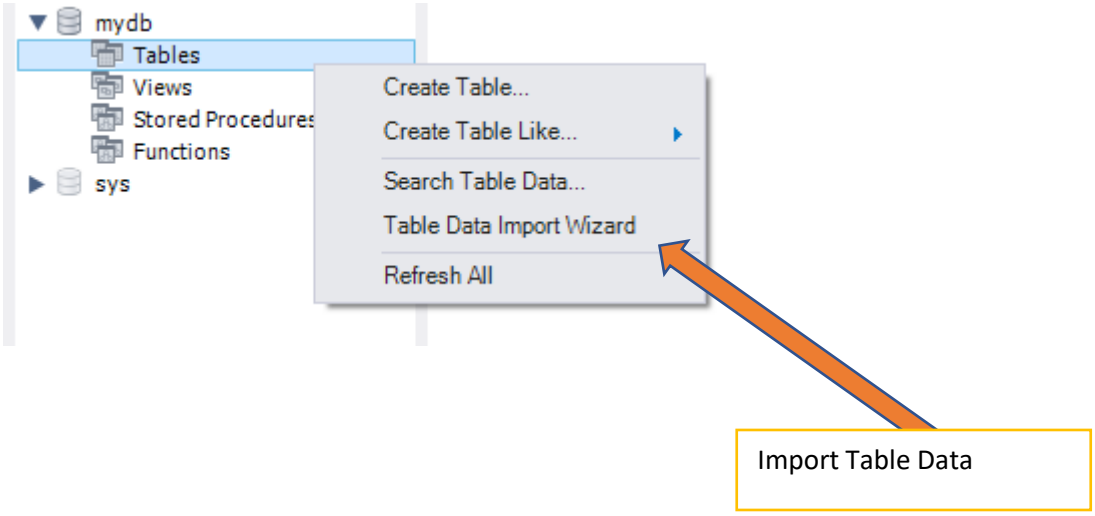
3. The connection is tested:



4.



5. Hacemos click derecho en las tablas



6.

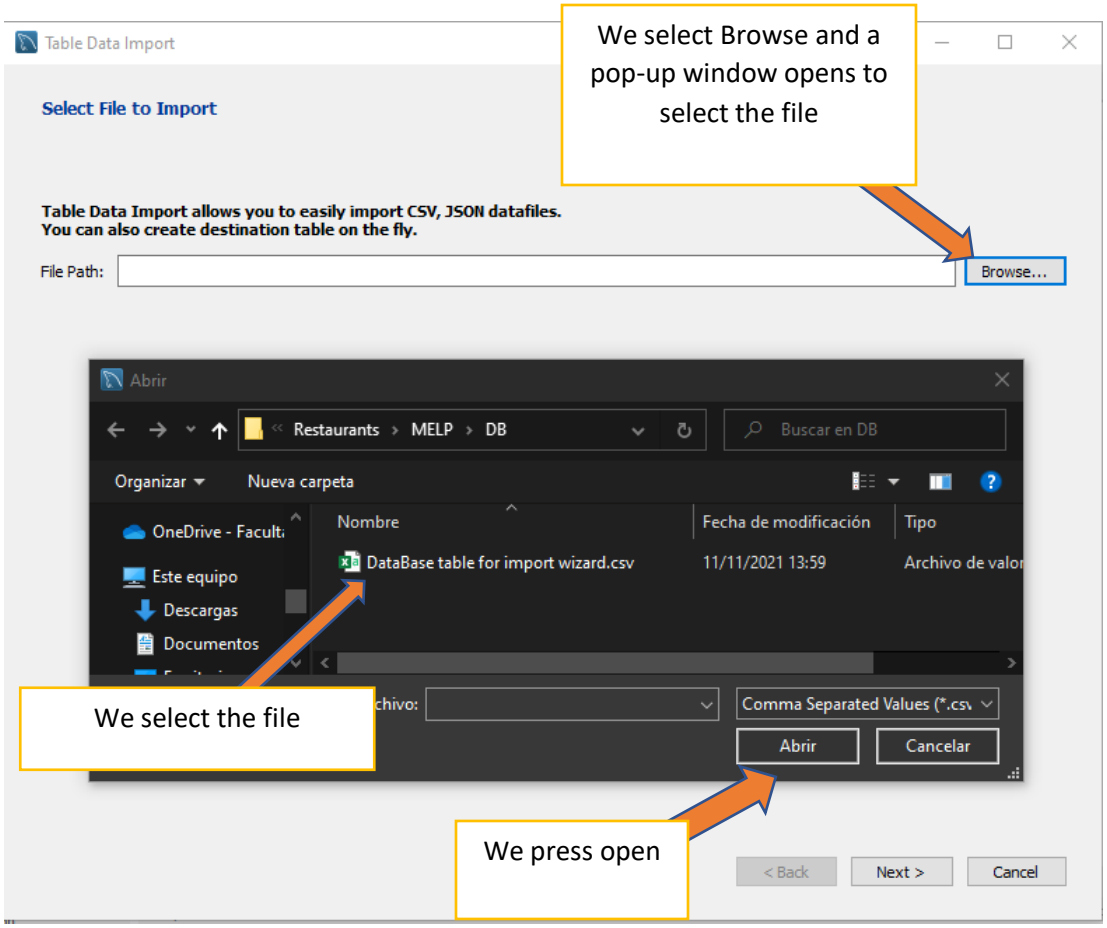


Table Data Import
Select File to Import
We select Browse and a pop-up window opens to select the file
Table Data Import allows you to easily import CSV, JSON datafiles.
You can also create destination table on the fly.
File Path:
Browse...
Abrir
Restaurants › MELP › DB
Buscar en DB
Organizar
Nueva carpeta
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
We select the file
Comma Separated Values (*.csv
Abrir
Cancelar
We press open
< Back
Next >
Cancel

7.

8.

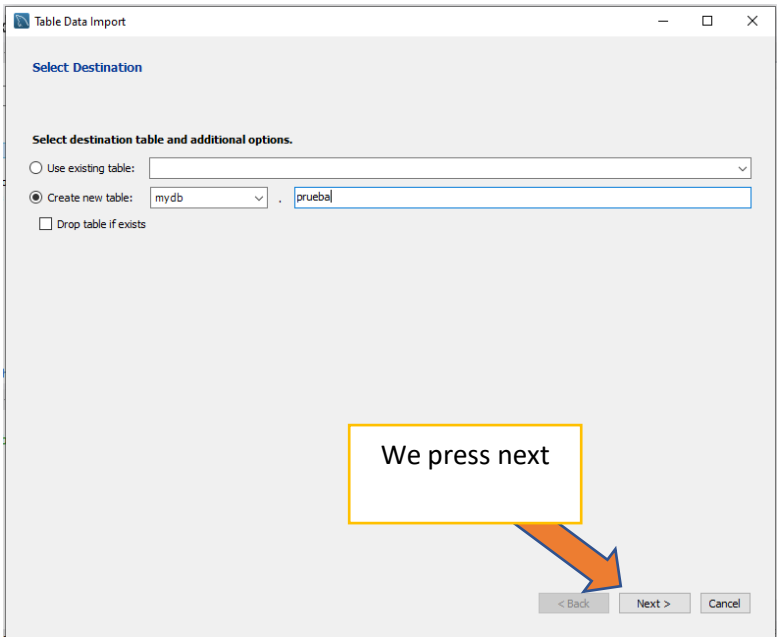
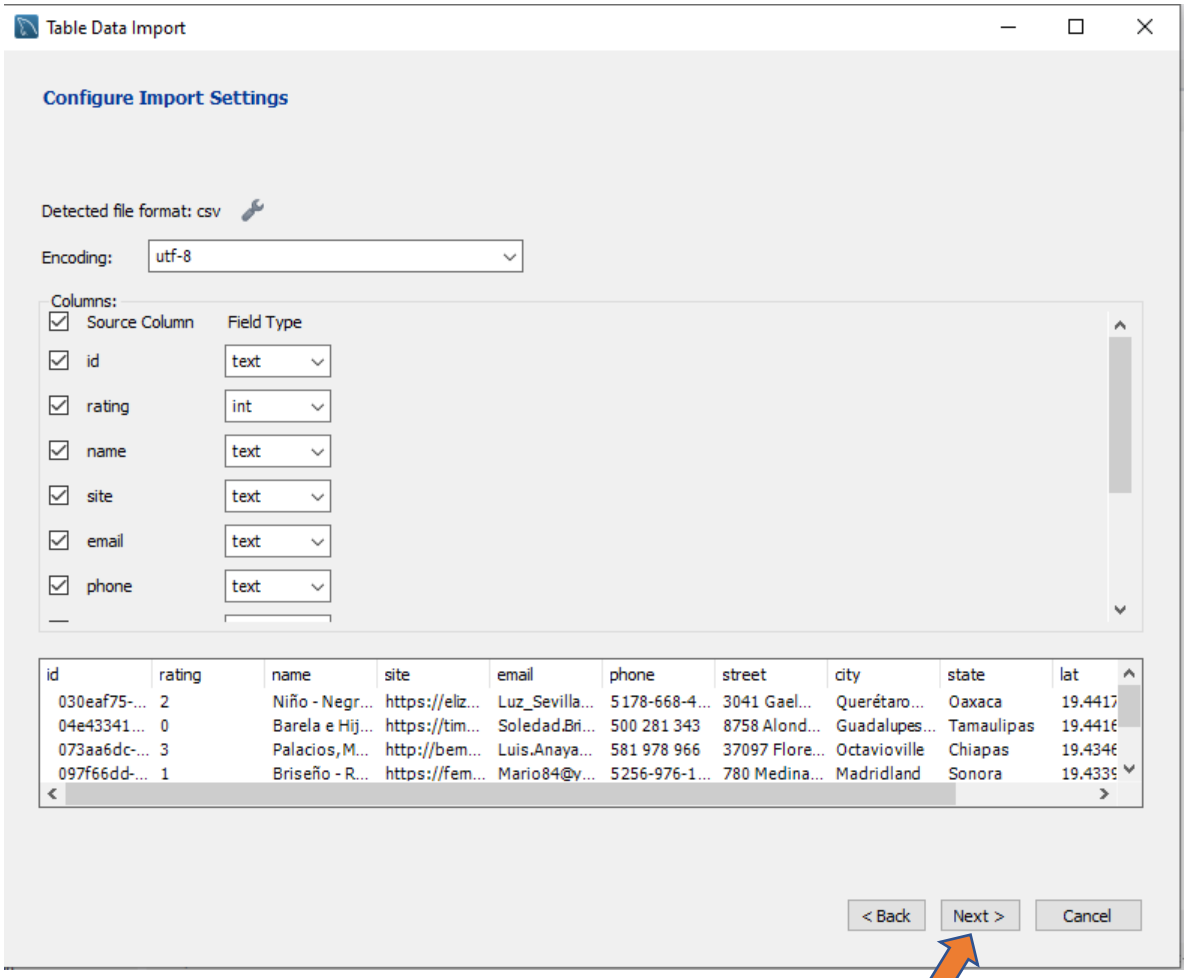


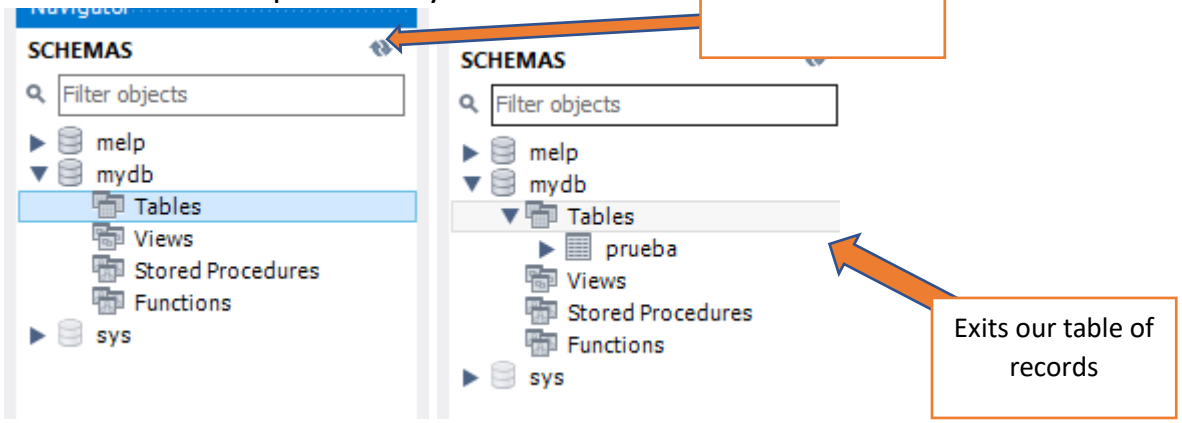
Table Data Import
Select Destination
Select destination table and additional options.
Use existing table:
Create new table:
mydb
prueba
Drop table if exists
We press next
< Back
Next >
Cancel

9.



We press next

10. Finalizamos la importación y actualizamos



We press next

Exits our table of records