PyMySQL

Para poder conectar nuestras aplicaciones Flask a una base de datos, necesitaremos un paquete que nos ayude. Utilizaremos uno llamado PyMySQL. A nivel de proyecto, ejecuta el siguiente comando en tu línea de comando para iniciar un nuevo proyecto.

new\_project\_folder $ pipenv install PyMySQL flask

Para cada proyecto que involucre una base de datos, realizaremos los siguientes pasos.

1. necesitaremos una base de datos a la que conectarnos. Creemos una en el entorno de trabajo llamada primer\_flask con una tabla única (mira el ERD a la derecha).
2. además del archivo server.py que creamos en cada proyecto, ahora también necesitaremos el siguiente archivo.

# un cursor es el objeto que usamos para interactuar con la base de datos

import pymysql.cursors

# esta clase nos dará una instancia de una conexión a nuestra base de datos

class MySQLConnection:

    def \_\_init\_\_(self, db):

        # cambiar el usuario y la contraseña según sea necesario DATOS DE MI CUENTA BASICAMENTE DE LARAGON QUEDA VACIO

        connection = pymysql.connect(

            host='localhost',

            user='root',

            password='root',

            db=db,

            charset='utf8mb4',

            cursorclass=pymysql.cursors.DictCursor,

            autocommit=True

        )

        # establecer la conexión a la base de datos

        self.connection = connection

    # el método para consultar la base de datos

    def query\_db(self, query, data=None):

        with self.connection.cursor() as cursor:

            try:

                query = cursor.mogrify(query, data)

                print("Running Query:", query)

                cursor.execute(query, data)

                if query.lower().find("insert") >= 0:

                    # las consultas INSERT devolverán el NÚMERO DE ID de la fila insertada

                    self.connection.commit()

                    return cursor.lastrowid

                elif query.lower().find("select") >= 0:

                    # las consultas SELECT devolverán los datos de la base de datos como una LISTA DE DICCIONARIOS

                    result = cursor.fetchall()

                    return result

                else:

                    # las consultas UPDATE y DELETE no devolverán nada

                    self.connection.commit()

            except Exception as e:

                # si la consulta falla, el método devolverá FALSE

                print("Something went wrong", e)

                return False

            finally:

                # cerrar la conexión

                self.connection.close()

# connectToMySQL recibe la base de datos que estamos usando y la usa para crear una instancia de MySQLConnection

def connectToMySQL(db):

    return MySQLConnection(db)

RESUMEN DEL CODIGO DE ARRIBA

* Las consultas SELECT devolverán una lista de diccionarios
* Las consultas INSERT devolverán el id generado automáticamente de la fila insertada
* Las consultas UPDATE y DELETE no devolverán nada
* Si la consulta falla, mal, devolverá False

1. Crear una clase que se asocie a la tabla con la que voy a trabajar. En el ejemplo de coding dojo:

first\_flask\_mysql/friend.py

# importar la función que devolverá una instancia de una conexión

from mysqlconnection import connectToMySQL

# modelar la clase después de la tabla friend de nuestra base de datos

class Friend:

    def \_\_init\_\_(self, data):

        self.id = data['id']

        self.first\_name = data['first\_name']

        self.last\_name = data['last\_name']

        self.occupation = data['occupation']

        self.created\_at = data['created\_at']

        self.updated\_at = data['updated\_at']

    # ahora usamos métodos de clase para consultar nuestra base de datos

    @classmethod

    def get\_all(cls):

        query = "SELECT \* FROM friends;"

        # asegúrate de llamar a la función connectToMySQL con el esquema al que te diriges

        results = connectToMySQL('first\_flask').query\_db(query) EL PROFE DIJO ESENCIAL

        # crear una lista vacía para agregar nuestras instancias de friends

        friends = []

        # Iterar sobre los resultados de la base de datos y crear instancias de friends con cls

        for friend in results:

            friends.append(cls(friend))

        return friends

1. Ahora vamos a actualizar nuestro archivo server.py para importar la clase y llamar al método de clase para consultar nuestra base de datos.

***first\_flask\_mysql/server.py***

    from flask import Flask, render\_template

# importar la clase de friend.py

from friend import Friend

app = Flask(\_\_name\_\_)

@app.route("/")

def index():

    # llamar al método de clase get all para obtener todos los amigos

    friends = Friend.get\_all()

    print(friends)

    return render\_template("index.html")

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

**Recuperar y visualizar datos**

Cuando el usuario visite la ruta raíz, queremos mostrar todos los amigos, por lo que nuestra lógica incluirá buscar a todos los amigos de la base de datos y luego representar esos datos en nuestra plantilla.

***first\_flask\_mysql/server.py***

    @app.route('/')

def index():

    friends = Friend.get\_all()

    print(friends)

    return render\_template("index.html", all\_friends = friends)

***first\_flask\_mysql/templates/index.html***

<h1>Todos mis amigos</h1>

{% for one\_friend in all\_friends %}

    <p>First Name: {{one\_friend.first\_name}}</p>

    <p>Last Name: {{one\_friend.last\_name}}</p>

    <p>Occupation: {{one\_friend.occupation}}</p>

    <hr>

{% endfor %}

Con nuestra sentencia SELECT \* FROM friends , estamos pidiendo todo lo que hay en la base de datos. Sin embargo, a menudo querremos proporcionar datos variables en la consulta que serían terribles de codificar en la consulta. Por ejemplo:

* SELECT \* FROM friends WHERE id=1  (donde el número de id real variará)
* UPDATE friends SET first\_name="Bryanna" WHERE id=9 (donde el nombre y la identificación reales variarán)

**Siempre que ejecutamos una consulta que incluye variables, debemos usar una sentencia preparada en lugar de la interpolación de cadenas.** En otras palabras, utiliza el siguiente patrón en lugar de cadenas f o la concatenación de cadenas.

Prácticamente, lo que esto significa es que necesitaremos una **variable de cadena para la consulta** y luego un **diccionario para los valores que se utilizarán en la cadena**. Cuando llamamos a la conexión de la base de datos para ejecutar la consulta, pasaremos tanto la consulta como el diccionario, así:

*Para evitar que el usuario acceda a los datos de mi db, debo crear la consulta y una variable de consulta, entonces pasa por una especie de filtro que protege mi base de datos.*

query = "UPDATE friends SET first\_name=%(fn)s WHERE id=%(id\_num)s;"

data= {

    "fn": #possibly a value from a form;

    "id\_num": #possibly a value from the url

}

mysql.query\_db(query, data)

1. **query = "UPDATE friends SET first\_name=%(fn)s WHERE id=%(id\_num)s;"**: Esta línea define una consulta SQL de actualización. La consulta actualiza la columna **first\_name** en la tabla **friends** donde el **id** coincide con un valor proporcionado.
2. **data = { "fn": #posiblemente un valor de un formulario, "id\_num": #posiblemente un valor de la URL }**: Aquí se define un diccionario llamado **data**. Este diccionario contiene dos pares clave-valor. La clave **"fn"** representa el valor que se utilizará para actualizar la columna **first\_name**, y la clave **"id\_num"** representa el valor que se utilizará para encontrar la fila específica que se actualizará.
3. **mysql.query\_db(query, data)**: Esta línea ejecuta la consulta SQL utilizando la función **query\_db** de la instancia de la conexión a la base de datos MySQL llamada **mysql**. La consulta se ejecuta utilizando los valores proporcionados en el diccionario **data**. En este caso, **%(fn)s** y **%(id\_num)s** son marcadores de posición que serán reemplazados por los valores correspondientes del diccionario **data**.

TODO LO QUE SEA CONSULTA SQL DEBE TENER % Y S FUERA DEL PARENTESIS SI SE TRATA DE UNA VARIABLE, LO QUE VA DENTRO DEL PARENTESIS ES EL NOMBRE DE LA VARIABLE.

Y AFUERA HAY UN DICCIONARIO DE DATOS: LOS VALORES QUE SE INTERPOLARÁN EN LA CADENA DE CONSULTA. CLAVES DEL DICCIONARIO DE DATOS - FN, ID\_NUM: LAS CLAVES DEL DICCIONARIO DE DATOS UTILIZADAS EN LA CADENA DE CONSULTA CON %-INTERPOLACIÓN

query = "SELECT \* FROM users WHERE email = %(email)s;"

data = { 'email' : request.form['email'] }

result = mysql.query\_db(query, data)

Aquí, estamos usando una **sentencia preparada** para crear nuestra consulta SQL con los datos proporcionados por nuestro usuario. Esto se hace dejando marcadores de posición en nuestra consulta que se completan con los valores de nuestro diccionario de datos.

# este código es solo para fines de demostración

# NO use este código en producción, TE dejará vulnerable a la inyección SQL

query = f"SELECT \* FROM users WHERE email = '{request.form['email']}';"

result = mysql.query\_db(query)

Al usar la interpolación de cadenas simple, esto convertiría nuestra consulta en:

SELECT \* FROM users WHERE email = 'joe@gmail.com' OR '1'='1'

Dado que '1' = '1' siempre se evaluará como verdadero, esta consulta ahora obtendrá todos los datos de la tabla de usuarios. Es posible que hayamos abierto una gran parte de nuestra base de datos a un usuario malintencionado. Cualquier usuario con conocimientos de SQL puede descubrir fácilmente cómo manipular nuestras consultas SQL. Pueden obtener acceso a datos confidenciales u obligarnos a ejecutar una consulta muy peligrosa.

**Otros ejemplos de inyecciones SQL**

Considera otro escenario en el que el usuario ingresó lo siguiente como su correo electrónico:

joe@gmail.com"; DROP TABLE users;

¿Qué hubiera pasado si la forma en que preparaste la consulta SQL fuera así?

query = f"SELECT \* FROM users WHERE email = '{request.form['email']}';"

result = mysql.query\_db(query)

La consulta que habría ejecutado habría sido

SELECT \* FROM users WHERE email = "joe@gmail.com"; DROP users;

¿Qué pasa si el usuario pasó lo siguiente como entrada de correo electrónico?

joe@gmail.com"; UPDATE users SET password = '\_\_\_\_' WHERE id = '\_\_\_'

Esto habría cambiado la contraseña de alguien. De manera similar, puedes ver cómo uno podría configurarse como administrador o recuperar información confidencial de la tabla de otros usuarios (por ejemplo, tarjeta de crédito, dirección, etc.). Las posibilidades son infinitas.

***Sentencias preparadas al rescate***

query = "SELECT \* FROM users WHERE email = %(email)s;"

# la variable de marcador de posición se llama email

# debe coincidir con el nombre de la clave en el diccionario de datos

data = {

    # esta clave de 'email' en los datos debe tener un nombre que coincida con el marcador en la consulta

    'email' : request.form['email']

}

result = mysql.query\_db(query, data)

Ahora que podemos recuperar amigos, agreguemos la funcionalidad que nos permite crear un amigo. Primero necesitamos crear un **@classmethod** para crear una fila en la base de datos.

***first\_flask\_mysql/friend.py***

# importar la función que devolverá una instancia de una conexión

from mysqlconnection import connectToMySQL

# modelar la clase después de la tabla friend de nuestra base de datos

class Friend:

    # ...otros métodos de clase

    # método de clase para guardar a nuestro amigo en la base de datos

    @classmethod

    def save(cls, data ):

        query = "INSERT INTO friends ( first\_name , last\_name , occupation , created\_at, updated\_at ) VALUES ( %(fname)s , %(lname)s , %(occ)s , NOW() , NOW() );"

        # data es un diccionario que se pasará al método de guardar desde server.py

        return connectToMySQL('first\_flask').query\_db( query, data )

A continuación, agreguemos un formulario a nuestra plantilla:

***first\_flask\_mysql/templates/index.html***

<h1>Agregar un Amigo</h1>

<form action="/create\_friend" method="POST">

    <label for="fname">First Name:</label>

    <input type="text" name="fname">

    <label for="lname">Last Name:</label>

    <input type="text" name="lname">

    <label for="occ" >Occupation:</label>

    <input type="text" name="occ">

    <input type="submit" value="Add Friend">

</form>

Ahora necesitamos un método para manejar el envío de nuestro formulario. Actualicemos nuestro archivo **server.py** y pasemos request.form al método save de la clase Friend.

***first\_flask\_mysql/server.py***

# fragmento de código relevante de server.py

from friend import Friend

@app.route('/create\_friend', methods=["POST"])

def create\_friend():

    # Primero hacemos un diccionario de datos a partir de nuestro request.form proveniente de nuestra plantilla

    # Las claves en los datos tienen que alinearse exactamente con las variables en nuestra cadena de consulta

    data = {

        "fname": request.form["fname"],

        "lname" : request.form["lname"],

        "occ" : request.form["occ"]

    }

    # Pasamos el diccionario de datos al método save de la clase Friend

    Friend.save(data)

    # No olvides redirigir después de guardar en la base de datos

    return redirect('/')

Nota: Hemos configurado el método query\_db para que cada consulta probada se imprima en la terminal. *Siempre que la consulta que elaboraste parece no funcionar o muestra un mensaje de error, investiga la consulta real que se está ejecutando en la terminal. Puedes intentar copiar y pegar la consulta en MySQL Workbench para ver si tienes la sintaxis correcta.*

Modelo-vista-controlador (MVC)

Los desarrolladores que diseñan entornos de trabajo tienen que tomar decisiones sobre la organización del código. Uno de los patrones más populares para organizar código es uno conocido como MVC: **Modelo-vista-controlador**. Este es un desglose básico de responsabilidades:

|  |  |
| --- | --- |
| **Modelo** | 1. Puede construir tablas de bases de datos 2. Maneja la lógica que se basa en datos 3. Se interrelaciona con la base de datos |
| **Vista** | 1. Página HTML que se sirve al cliente 2. Puede contener cierta lógica para ser manejada por un motor de plantillas |
| **Controlador** | 1. Recibe solicitudes entrantes 2. Lógica mínima 3. Llama modelos para agregar/procesar datos 4. Determina la respuesta adecuada |

Paso 1: la carpeta de la aplicación

El proceso

1. Crea un directorio llamado **flask\_app**
2. Crea el archivo **\_\_init\_\_.py** dentro de la carpeta **flask\_app**
3. Inserta este código:

# \_\_init\_\_.py

from flask import Flask

app = Flask(\_\_name\_\_)

app.secret\_key = "shhhhhh"copy

1. Elimina las líneas anteriores de **server.py**
2. En **server.py** agregamos esta línea:

from flask\_app import app

# ...server.py

copy

1. Mover la carpeta templates a **flask\_app**
2. Mover la carpeta estática a **flask\_app**