

## Práctica 06. API Rest en Flask

Administracion y Diseño de Bases de Datos

Cheuk Kelly Ng Pante (alu0101364544@ull.edu.es)

12 de diciembre de 2023

# Índice general

|  |          |
|--|----------|
| <b>1. Introducción</b>   | <b>1</b> |
| <b>2. Actividad 1</b>  | <b>1</b> |
| 2.1. Instalación del framework <i>Flask</i> y la biblioteca <i>psycpg2-binary</i> . . . . .  | 1        |
| 2.2. Despliegue de la aplicación web . . . . .   | 1        |
| 2.3. Creación de la base de datos e inserción de datos . . . . .   | 2        |
| 2.4. Personalizar la referencia <i>About</i> . . . . .   | 2        |
| 2.5. Verificar funcionamiento de la operación de visualizar los registros . . . . .  | 2        |
| 2.6. Verificar operación de inserción de registros . . . . .   | 2        |
| 2.7. Construcción de una REST API para la operación de borrado . . . . .   | 2        |
| 2.8. Construcción de una REST API para la operación de actualización . . . . .   | 3        |
| <b>3. Actividad 2</b>  | <b>5</b> |
| 3.1. Desplegar la base de datos <i>MyHome.psql</i> . . . . .   | 5        |
| 3.2. Construir con <i>Flask</i> y <i>psycpg2</i> las siguientes REST API . . . . .   | 5        |
| 3.2.1. Retorno de la temperatura media de todas las habitaciones . . . . .   | 5        |
| 3.2.2. Retorno de la temperatura máxima de una habitación . . . . .  | 6        |
| 3.2.3. Dado el <i>room_id</i> retornar el nombre de la habitación . . . . .  | 6        |
| 3.2.4. Dado el <i>room_id</i> retornar la temperatura media histórica de la habitación. . . . .  | 6        |
| 3.2.5. Dado el <i>room_id</i> retornar la temperatura mínima en la habitación y el nombre la habitación en formato <i>JSON</i> . . . . . | 6        |
| <b>4. Chequeo de excepciones</b>   | <b>7</b> |

## 1. Introducción

La API RESTful es una interfaz que dos sistemas de computación utilizan para intercambiar información de manera segura a través de Internet. La mayoría de las aplicaciones para empresas deben comunicarse con otras aplicaciones internas o de terceros para llevar a cabo varias tareas

Flask es un framework para desarrollo web escrito en Python. Se puede utilizar para diversos tipos de aplicación, entre ellas, desarrollo de APIs. Existen muchas maneras de implementar un API REST en Flask.

## 2. Actividad 1

### 2.1. Instalación del framework *Flask* y la biblioteca *psycpg2-binary*

Para la instalación de *Flask* y *psycpg2-binary*, así como el desarrollo de la práctica se va a crear un entorno virtual con *virtualenv*. Primero se crea un directorio para el entorno virtual y se accede a él.

```
$ mkdir actividad_1
$ cd actividad_1
```

Dentro del directorio se crea el entorno virtual con *virtualenv* y se instalará *Flask* y *psycpg2-binary* dentro de él. Para ello, se ejecutarán los siguientes comandos:

```
$ sudo apt install python3.10-venv
$ python3 -m venv venv
$ . venv/bin/activate
$ pip install Flask
$ pip install psycpg2-binary
```

### 2.2. Despliegue de la aplicación web

Para desplegar la aplicación web se va a crear una base de datos en *PostgreSQL* con el nombre de *flask\_db*. Para ello, ejecutamos dentro *psql* el siguiente comando:

```
postgres=# CREATE DATABASE flask_db;
```

Una vez creada la base de datos, sobre el directorio *actividad\_1* se ponen los archivos que se van a utilizar para el desarrollo de la práctica. Estos archivos son: *app.py* e *init\_db.py* y dentro de estos ficheros añadimos el usuario y contraseña de *postgres* para poder acceder a la base de datos.

Ya modificados los archivos, se ejecuta el siguiente comando para desplegar la aplicación web:

```
$ python3 app.py
```

También se puede ejecutar el siguiente comando para desplegar la aplicación web:

```
$ flask --app app.py run --host 0.0.0.0 --port=8080
```

Con los comandos anteriores desplegamos la aplicación en local en el puerto 8080, pero va a fallar ya que antes necesita la inicialización de la base de datos, lo cual se hace con el siguiente comando:

```
$ python3 init_db.py
```

## 2.3. Creación de la base de datos e inserción de datos

La creación de la base de datos se hizo en el apartado anterior y la inserción de datos se crean en el script *init\_db.py* y se han realizado nueve inserciones en la tabla *books*.

## 2.4. Personalizar la referencia *About*

Para personalizar la referencia del about creamos un nuevo archivo html en el directorio templates *about.html* en donde añadimos los nombres y apellidos de los integrantes del grupo. Luego modificamos *base.html* y el fichero *app.py* para poder acceder a la nueva sección.

## 2.5. Verificar funcionamiento de la operación de visualizar los registros

Se puede verificar que se pueden visualizar los registros de la base de datos gracias a la función *index()* que se encuentra en el fichero *app.py*, en el que se realiza la siguiente consulta para conseguir todas las entradas de la tabla *books*:

```
1 SELECT *
2 FROM books;
```

## 2.6. Verificar operacion de inserción de registros

Se puede verificar que se pueden insertar registros en la base de datos gracias a la función *create()* que se encuentra en el fichero *app.py*, en el que se realiza la siguiente consulta para insertar una entrada en la tabla *books*:

```
1 INSERT INTO books (title, author, pages_num, review)
2 VALUES (%s, %s, %s, %s);
```

## 2.7. Construcción de una REST API para la operación de borrado

Para construir una REST API para la operación de borrado se crea una nueva función en el fichero *app.py* llamada *delete()*, aquí el código en python:

```
1 @app.route('/delete/', methods=['GET', 'POST'])
2 def delete():
3     if request.method == 'POST':
4         id = request.form['ID']
5
6         conn = get_db_connection()
7         cur = conn.cursor()
8         cur.execute('DELETE FROM books WHERE id = %s',
9                     (id))
10        conn.commit()
11        cur.close()
12        conn.close()
13        return redirect(url_for('index'))
14
15    return render_template('delete.html')
```

Este código en python lo que hace es una consulta en sql para borrar una entrada de la tabla *books*, aquí el código en sql:

```
1 DELETE FROM books
2 WHERE id = %s;
```

Además, hay que modificar el fichero *base.html* para crear un apartado en el que se pueda borrar un registro de la base de datos. Creamos fichero *delete.html* en el directorio *templates* y lo codificamos para que se pueda borrar un registro de la base de datos con un formulario, aquí el código en html:

```

1 {% extends 'base.html' %}
2
3 {% block content %}
4 <h1>{% block title %} Delete a Book {% endblock %}</h1>
5 <form method="POST">
6     <p>
7         <label for="ID">ID</label>
8         <input type="text" name="ID" placeholder="Book ID">
9     </input>
10 </p>
11 <p>
12     <button type="submit">Submit</button>
13 </p>
14 </form>
15 {% endblock %}

```

## 2.8. Construcción de una REST API para la operación de actualización

Para construir una REST API para la operación de actualización se crea una nueva función en el fichero *app.py* llamada *update()*, en el que se actualizará un registro indicando su ID, aquí el código en python:

```

1 @app.route('/update/', methods=['GET', 'POST'])
2 def update():
3     if request.method == 'POST':
4         id = request.form['ID']
5         title = request.form['title']
6         author = request.form['author']
7         review = request.form['review']
8
9         conn = get_db_connection()
10        cur = conn.cursor()
11        cur.execute('UPDATE books SET title = %s, author = %s, review = %s WHERE id = %s
12        ,
13                    (title, author, review, id))
14        conn.commit()
15        cur.close()
16        conn.close()
17        return redirect(url_for('index'))
18
19    return render_template('update.html')

```

Este código en python lo que hace es una consulta en sql para actualizar una entrada de la tabla *books*, aquí el código en sql:

```

1 UPDATE books
2 SET title = %s, author = %s, review = %s WHERE id = %s;

```

Luego, hay que modificar el fichero *base.html* para crear un apartado en el que se pueda actualizar un registro de la base de datos. Creamos fichero *update.html* en el directorio *templates* y lo codificamos para que se pueda actualizar un registro de la base de datos con un formulario, aquí el código en html:

```

1 {% extends 'base.html' %}
2
3 {% block content %}
4 <h1>{% block title %} Update a Book {% endblock %}</h1>
5 <form method="post">
6   <p>
7     <label for="ID">ID</label>
8     <input type="text" name="ID" placeholder="Book ID">
9   </input>
10 </p>
11
12 <p>
13   <label for="title">Title</label>
14   <input type="text" name="title" placeholder="Book title">
15 </input>
16 </p>
17
18 <p>
19   <label for="author">Author</label>
20   <input type="text" name="author" placeholder="Book author">
21 </input>
22 </p>
23
24 <p>
25   <label for="pages_num">Number of pages</label>
26   <input type="number" name="pages_num" placeholder="Number of pages">
27 </input>
28 </p>
29
30 <p>
31   <label for="review">Review</label>
32   <br>
33   <textarea name="review" placeholder="Review" rows="15" cols="60"></textarea>
34 </p>
35 <p>
36   <button type="submit">Submit</button>
37 </p>
38 </form>
39 {% endblock %}

```

## 3. Actividad 2

### 3.1. Desplegar la base de datos *MyHome.psql*

Para desplegar la base de datos a partir del script *MyHome.sql* se ha utilizado el siguiente comando:

```
postgres=# \i MyHome.psql
```

### 3.2. Construir con *Flask* y *psycopg2* las siguientes REST API

Se va a crear un nuevo directorio de trabajo llamado *actividad.2* en el que tendrá las definiciones de la nueva aplicación, las funcionalidades estarán en el fichero *room\_app.py*.

#### 3.2.1. Retorno de la temperatura media de todas las habitaciones

Para retornar la temperatura media de todas las habitaciones se crea una nueva función en el fichero *room\_app.py* llamada *get\_avg\_temperature()*, en el que se retornará la temperatura media de todas las habitaciones, aquí el código en python:

```
1 @app.route('/average/', methods=('GET', 'POST'))
2 def average():
3     conn = get_db_connection()
4     cur = conn.cursor()
5     cur.execute('SELECT AVG(temperature)
6                 'FROM public.temperatures;')
7     values = cur.fetchall()
8     cur.close()
9     conn.close()
10    return render_template('average.html', rooms = values)
```

Lo que hace este código en python es una consulta en sql para obtener la temperatura media de todas las habitaciones,

```
1 SELECT AVG(temperature)
2 FROM temperatures;
```

Luego, hay que modificar el fichero *base.html* para crear un apartado en el que se pueda obtener la temperatura media de todas las habitaciones. Creamos fichero *average.html* en el directorio *templates* y lo codificamos para que se pueda obtener la temperatura media de todas las habitaciones, aquí el código en html:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4     <h1>{% block title %} Rooms {% endblock %}</h1>
5     {% for room in rooms %}
6         <div class='room'>
7             <h3>{{ room[0] }} Celsius</h3>
8         </div>
9     {% endfor %}
10 {% endblock %}
```

### 3.2.2. Retorno de la temperatura máxima de una habitación

En este apartado se va a retornar la temperatura máxima de una habitación, para ello se hará lo mismo que en el apartado anterior pero en la consulta extraemos la temperatura máxima de una habitación, aquí el código en SQL:

```
1 SELECT MAX(temperature)
2 FROM temperatures;
```

### 3.2.3. Dado el *room\_id* retornar el nombre de la habitación

Para retornar el nombre de la habitación dado el *room\_id* se crea una nueva función en el fichero *room\_app.py* llamada *room\_name()*, en el que se retornará el nombre de la habitación dado el *room\_id*, aquí el código en python:

```
1 @app.route('/name_room/', methods=('GET', 'POST'))
2 def name_room():
3     if request.method == 'POST':
4         id = request.form['ID']
5         conn = get_db_connection()
6         cur = conn.cursor()
7         cur.execute('SELECT name FROM public.rooms WHERE id = %s;', (id))
8         values = cur.fetchall()
9         cur.close()
10        conn.close()
11        return render_template('result_name_room.html', rooms=values)
12    return render_template('name_room.html')
```

Lo que hace este código en python es una consulta en sql para obtener el nombre de la habitación dado el *room\_id*, aquí el código en sql:

```
1 SELECT name
2 FROM public.rooms
3 WHERE id = %s;
```

Para este procedimiento se ha creados dos ficheros html, uno para mostrar el formulario y otro para mostrar el resultado.

### 3.2.4. Dado el *room\_id* retornar la temperatura media histórica de la habitación.

Para retornar la temperatura media histórica de la habitación dado el *room\_id* es similar al apartado anterior, se crea una nueva función en el fichero *room\_app.py* llamada *room\_avg\_temperature()*, en el que se retornará la temperatura media histórica de la habitación dado el *room\_id*, aquí el código en SQL:

```
1 SELECT AVG(temperature)
2 FROM temperatures
3 WHERE room_id = %s;
```

### 3.2.5. Dado el *room\_id* retornar la temperatura mínima en la habitación y el nombre la habitación en formato *JSON*

Esta REST API es parecido al apartado anterior, se crea una nueva función en el fichero *room\_app.py* llamada *room\_min\_temperature()*, en el que se retornará la temperatura mínima en la habitación y el nombre la habitación en formato *JSON*, aquí el código en SQL:



```

1 SELECT MIN(temperature), r.name
2 FROM public.rooms r
3 JOIN public.temperatures t ON r.id = t.room_id
4 WHERE room_id = 10
5 GROUP BY(room_id, name);

```

## 4. Chequeo de excepciones

Para el chequeo de excepciones se ha creado unas funciones en el script que son las siguientes:

- Manejo de errores 404:

```

1 # Manejo de error 404 - Pagina no encontrada
2 @app.errorhandler(404)
3 def page_not_found(error):
4     return render_template("404.html"), 404

```

- Manejo de errores 500:

```

1 # Manejo de error 500 - Error interno del servidor
2 @app.errorhandler(500)
3 def internal_server_error(error):
4     return render_template("500.html"), 500

```

- Manejo de errores 403:

```

1 # Ruta para generar un error 403 - Prohibido
2 @app.route("/forbidden")
3 def forbidden():
4     abort(403)

```

Luego dentro de cada función que hace la aplicación se ha añadido un try-except para manejar las excepciones y en todas las funciones se ha añadido el mismo código para manejar las excepciones aunque en algunas se ha añadido un código adicional dependiendo de las necesidades de la función, aquí el código en python resumiendo como fueron implementadas las excepciones

```

1 @app.route('/delete/', methods=['GET', 'POST'])
2 def delete():
3     try:
4         if request.method == 'POST':
5
6             #...codigo
7
8             return redirect(url_for('index'))
9
10            return render_template('delete.html')
11
12    except BadRequest as e:
13        print(f"Error 400: {str(e)}")
14        return render_template("400.html", error=str(e)), 400
15
16    except psycpg2.Error as e:
17        print(f"Error al eliminar un libro en la base de datos: {str(e)}")
18        raise InternalServerError(
19            "Error interno del servidor al eliminar un nuevo libro."

```

```
20     )
21
22     except NotFound as e:
23         print(f"Error 404: {str(e)}")
24         return render_template("404.html", error=e), 404
```

En el script *app.py* se puede ver el código completo de la aplicación con el chequeo de excepciones.