



**UNIVERSIDAD DE
SAN BUENAVENTURA**

Andres Felipe Lasso Perdomo

30000097453

andresfelipe.lasso@gmail.com

FizzBuzz web

Técnicas de programación avanzadas

2/05/2024

Problemática:

Un sistema web es un ejemplo clásico de una arquitectura cliente servidor, en la que es posible acceder a

recursos mediante una combinación de URLs y peticiones o verbos HTTP.

En este ejercicio se emplearán una serie de operaciones sobre un FizzBuzz Web, empleando Flask como

servidor liviano y Postman como herramienta de pruebas.

requisitos y restricciones:

Usuario estándar:

- Puede hacer registro (sign-up), acceso (log-in) y cierre de sesión (log-out).
- El servidor de FizzBuzz se apoya en una estrategia de persistencia, en la que inicia con 100 números y sus respectivos valores de FizzBuzz. La persistencia puede ser lograda mediante archivos o una base de datos, pero se espera que se aplique una estrategia de inversión de dependencias.
- Los 100 números iniciales corresponderán a los empleados en ejercicios previos y disponibles en Replit.
- Una petición GET para un número presente en el repositorio, y enviado como parte de la URL, retorna su respectivo valor de FizzBuzz y el código HTTP 200. La API sugerida es “fb”, pero se tiene la libertad de especificar otra ruta.
- Una petición GET para un número NO presente en el repositorio, y enviado como parte de la URL, retorna la cadena “Not Found” y el código HTTP 404.
- Una petición POST, para un número NO presente (o no activo) en el repositorio, y enviado como parte de la URL, agrega el número y su respectivo valor de FizzBuzz al repositorio, retorna el valor de FizzBuzz agregado, y el código HTTP 201. Si al número indicado, se le ha realizado un borrado lógico, este se reactivará, y se retornará el valor de FizzBuzz, acompañado del código HTTP 200.

- Una petición POST, para un número presente en el repositorio, y enviado como parte de la URL, NO

modifica el repositorio, retorna el valor de FizzBuzz respectivo, y el código HTTP 409.

- Una petición POST, para la ruta “range”, y un body indicando límite inferior y límite superior, retornará

todos los números y los valores de FizzBuzz en el intervalo [límite inferior, límite superior], acompañado

del código HTTP 200. En caso de que no existan valores en el intervalo, se retornará cadena “Not Found”

y el código HTTP 404. Se espera que no se emplee un algoritmo trivial o ingenuo para la verificación de

la intersección de los números presentes en el repositorio, con la base de datos.

- Una petición DELETE, acompañado de un número en la URL, para la ruta “fb”, y presente en el

repositorio, realizará un borrado lógico para el número, retornará la cadena “No Content” y el código

HTTP 204.

- Una petición DELETE, acompañado de un número en la URL, para la ruta “fb”, y NO presente en el

repositorio, retornará la cadena “Not Found” y el código HTTP 404.

- Cualquier otra petición, debe retornar un código HTTP 400.

Usuario administrador:

- Puede hacer acceso (log-in) y cierre de sesión (log-out).

- Una petición DELETE, acompañado de un número en la URL, para la ruta “fb”, y presente en el repositorio, realizará un borrado duro para el número, retornará la cadena “No Content” y el código HTTP 204.
- El resto de operaciones del usuario administrador, serán iguales a la del usuario estándar.

Pruebas:

Para las pruebas se utilizará postman. El código parte de la iteración anterior, así que todos los métodos de request ya fueron implementados y probados, por lo que no se harán pruebas de eso en esta iteración. Lo que se probará es la implementación de un log in, sign in, y un log out. Además una nueva funcionalidad en la que si un usuario administrador realiza una petición delete, se elimina el dato de la base de datos.

Se utilizarán las mismas pruebas de la iteración anterior, pero su análisis será el opuesto. Si sin necesidad de hacer un log in se pueden obtener los datos significa que las pruebas fallaron y el programa no implementa un mecanismo de autorización.

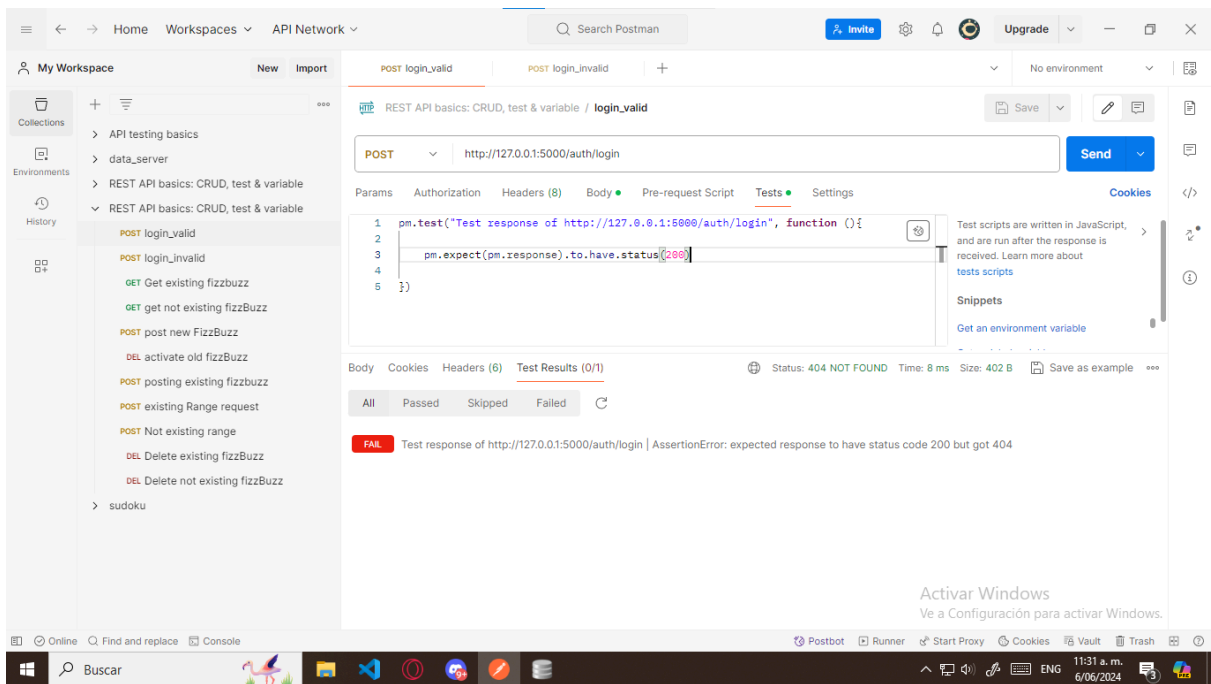
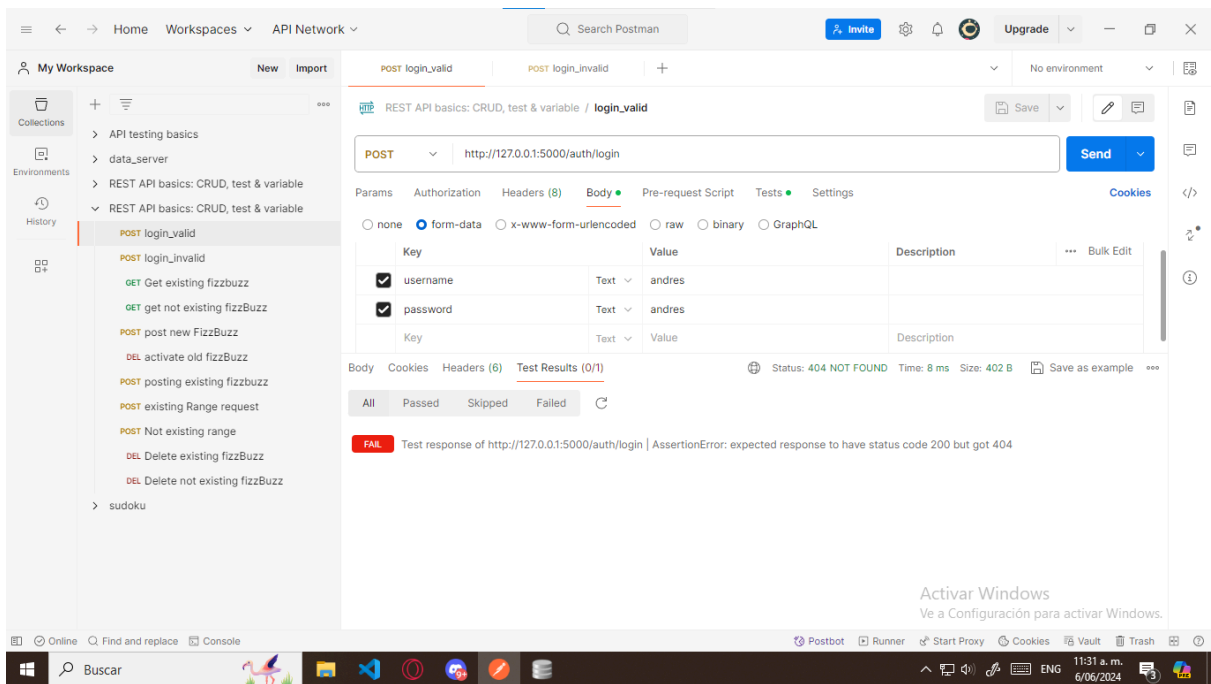
RED

The screenshot displays the REST Client application interface. The left sidebar shows a collection of API tests under 'My Workspace'. The main panel shows the 'Run results' for the selected test 'REST API basics: CRUD, test...'. The results table indicates that all tests passed successfully.

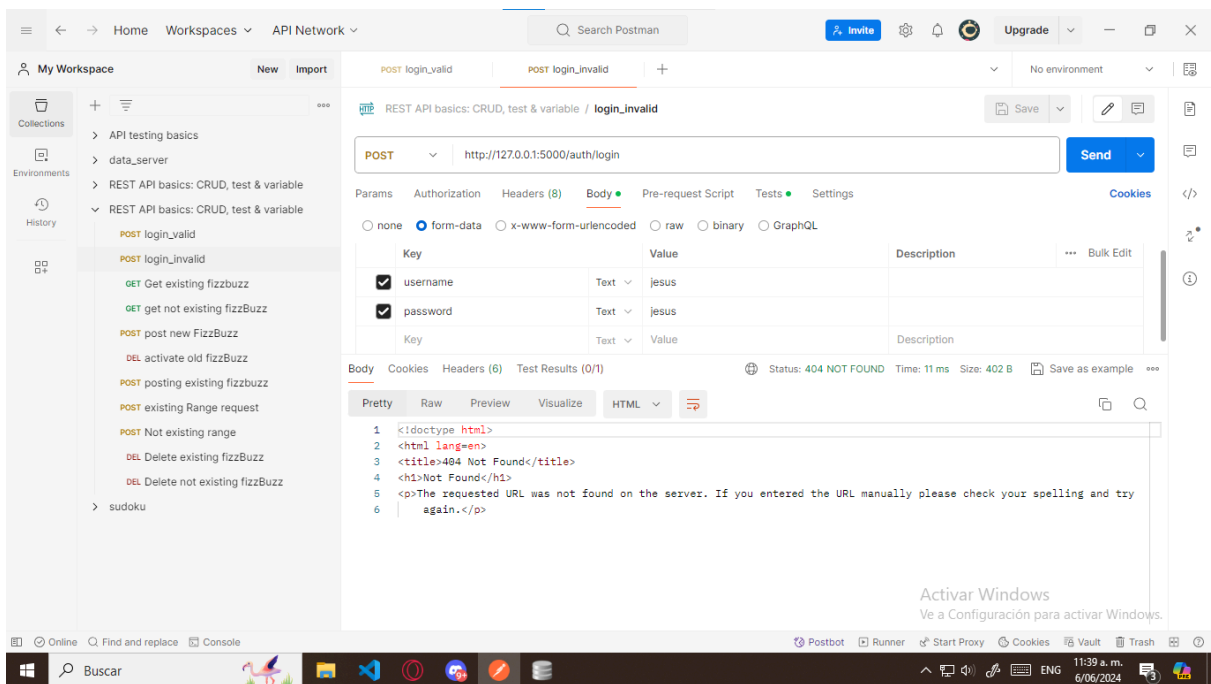
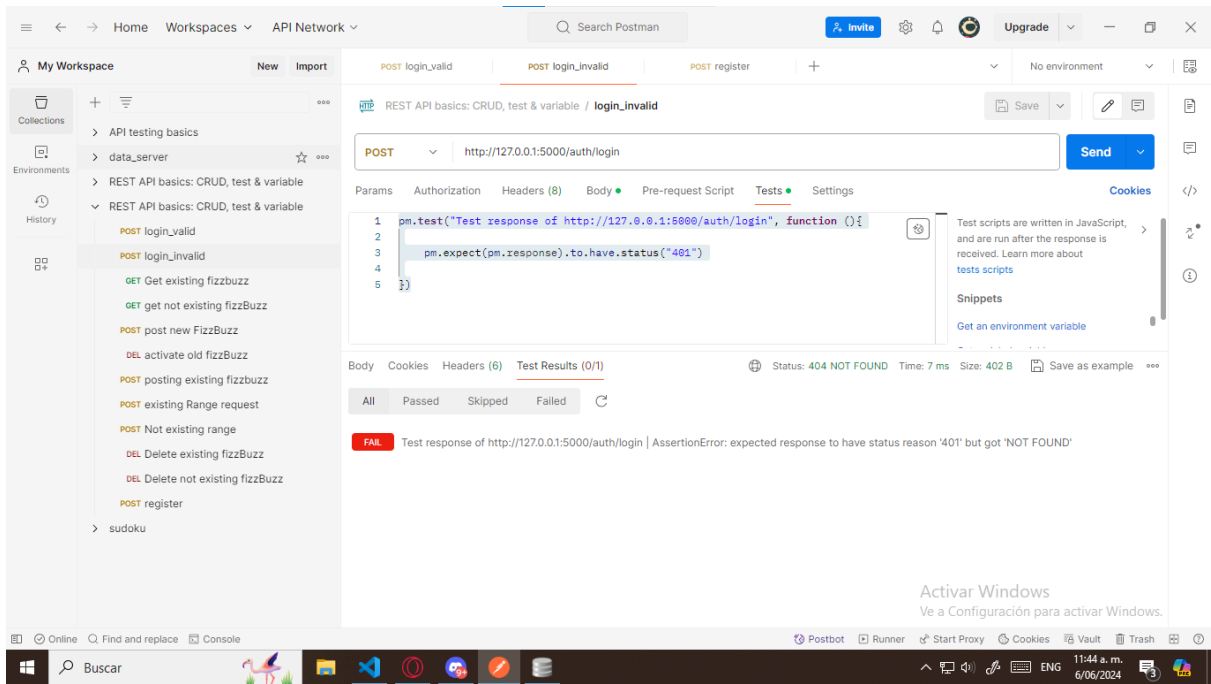
Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	6s 652ms	225	9 ms

Method	Test Name	Pass/Fail
GET	Get existing fizzbuzz	200 0
GET	get not existing fizzBuzz	2 0
POST	post new FizzBuzz	11 0
DELETE	activate old fizzBuzz	2 0
POST	posting existing fizzbuzz	2 0
POST	existing Range request	2 0
POST	Not existing range	2 0
DELETE	Delete existing fizzBuzz	1 1
DELETE	Delete not existing fizzBuzz	2 0
DELETE	login	0 0

Aquí se hicieron todas las solicitudes de la iteración anterior. y sin necesidad de mandar una request a login, todas las solicitudes se ejecutaron correctamente y se obtuvieron los datos. Esto significa que la prueba falla, y que no se está implementando un método de autorización.



En esta prueba se verifica si existe la ruta login y se le envía un usuario y contraseña válidos, de administrador. La prueba falla ya que no se ha implementado la ruta.



En esta prueba se ingresa a log in un usuario invalido, debería volver a la página de login y retorna un código http 401, como no está implementada esta ruta la prueba falla.

Home Workspaces API Network Search Postman

My Workspace New Import

REST API basics: CRUD, test & variable / register

POST http://127.0.0.1:5000/auth/register

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description
username	martin	
password	martin	

Body Cookies Headers (6) Test Results (0/1) Status: 404 NOT FOUND Time: 11 ms Size: 402 B Save as example

Pretty Raw Preview Visualize HTML

```
<!doctype html>
<html lang=en>
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.</p>
```

Activar Windows
Ve a Configuración para activar Windows.

Online Find and replace Console Postbot Runner Start Proxy Cookies Vault Trash

Buscar 11:44 a. m. 6/06/2024

Home Workspaces API Network Search Postman

My Workspace New Import

REST API basics: CRUD, test & variable / register

POST http://127.0.0.1:5000/auth/register

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

```
pm.test("Test response of http://127.0.0.1:5000/auth/register", function () {
  pm.expect(pm.response).to.have.status("200")
})
```

Test scripts are written in JavaScript, and are run after the response is received. Learn more about tests scripts

Snippets
Get an environment variable

Body Cookies Headers (6) Test Results (0/1) Status: 404 NOT FOUND Time: 11 ms Size: 402 B Save as example

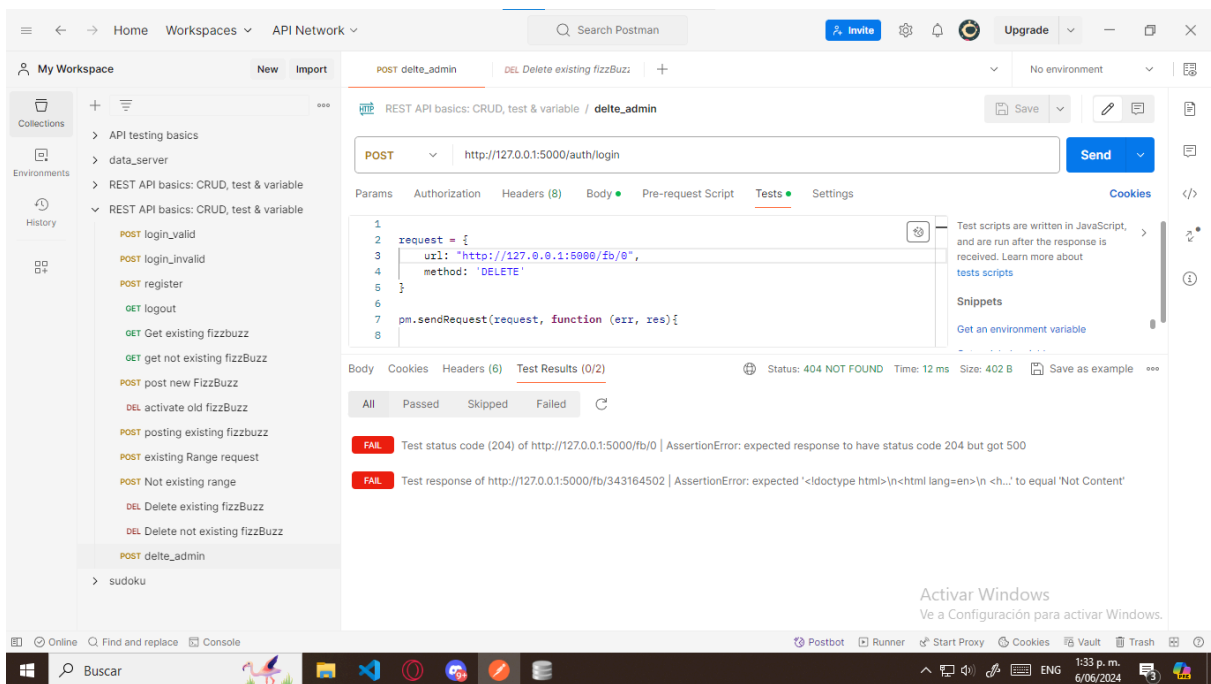
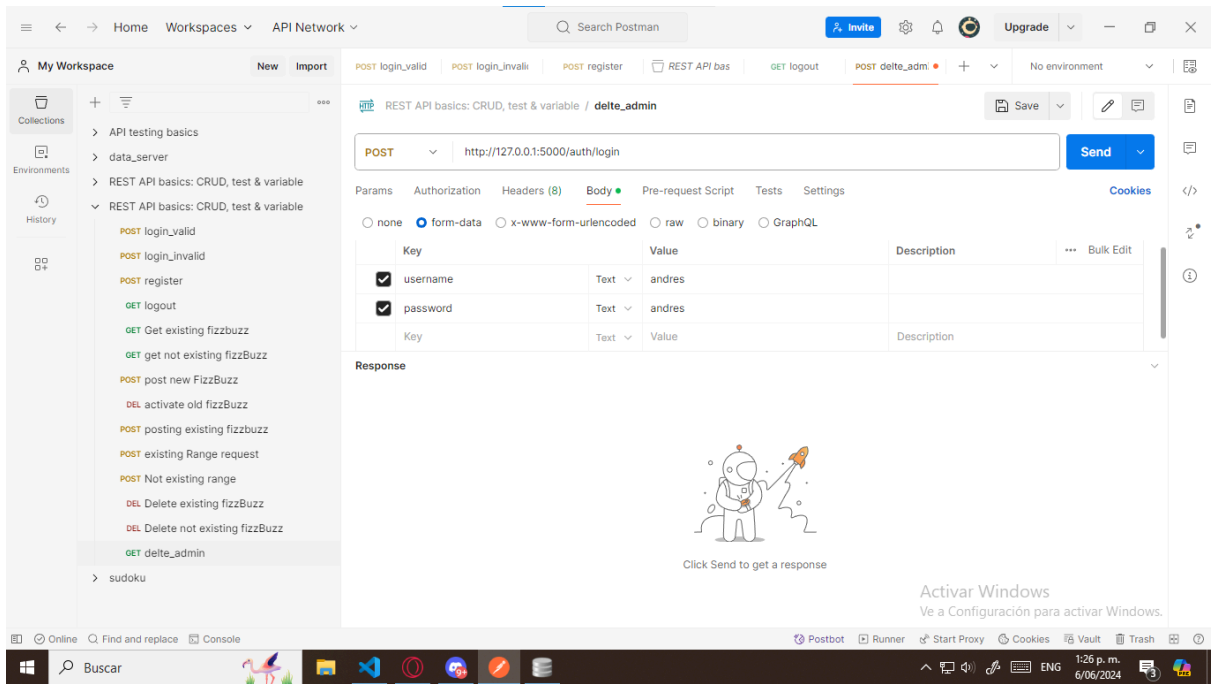
All Passed Skipped Failed

FAIL Test response of http://127.0.0.1:5000/auth/register | AssertionError: expected response to have status reason '200' but got 'NOT FOUND'

Activar Windows
Ve a Configuración para activar Windows.

Online Find and replace Console Postbot Runner Start Proxy Cookies Vault Trash

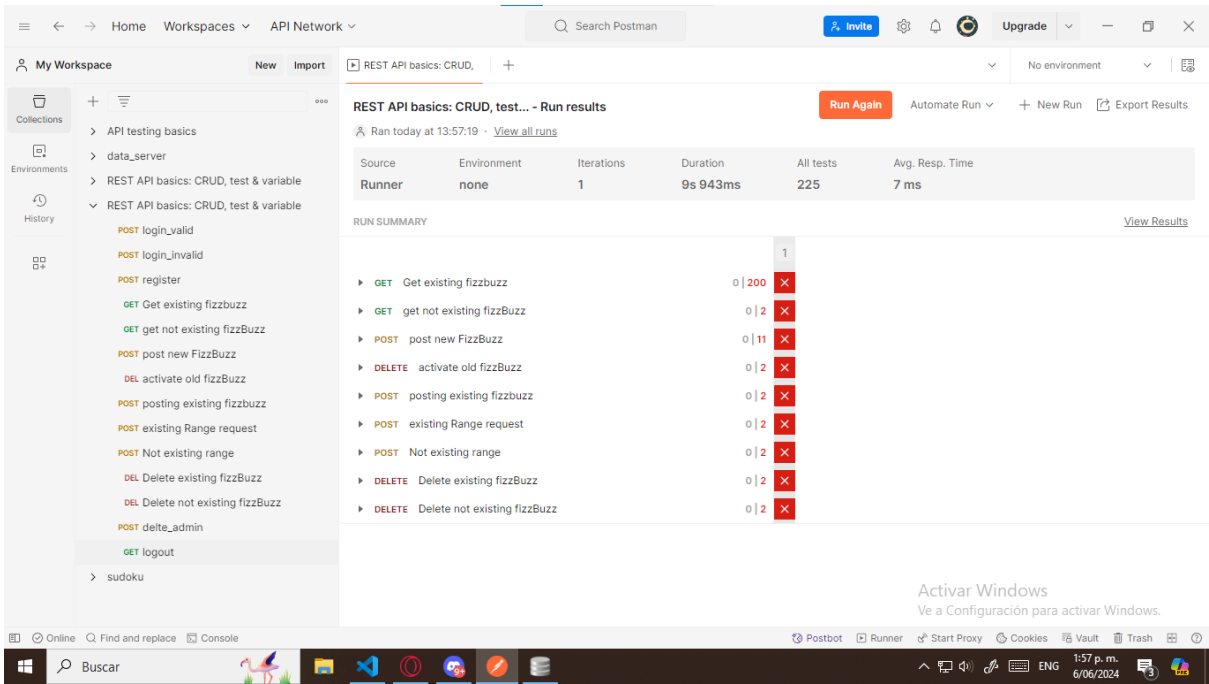
Buscar 11:44 a. m. 6/06/2024



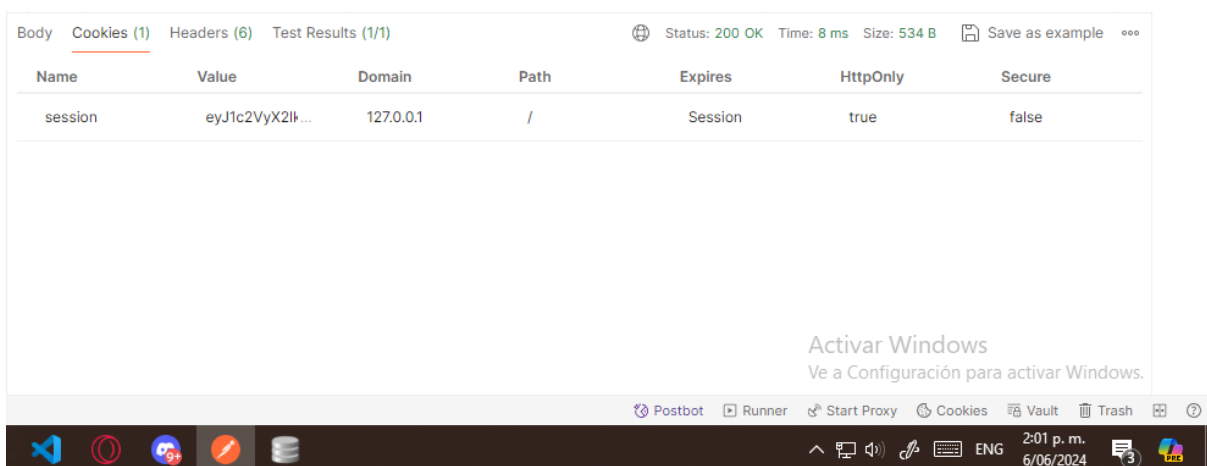
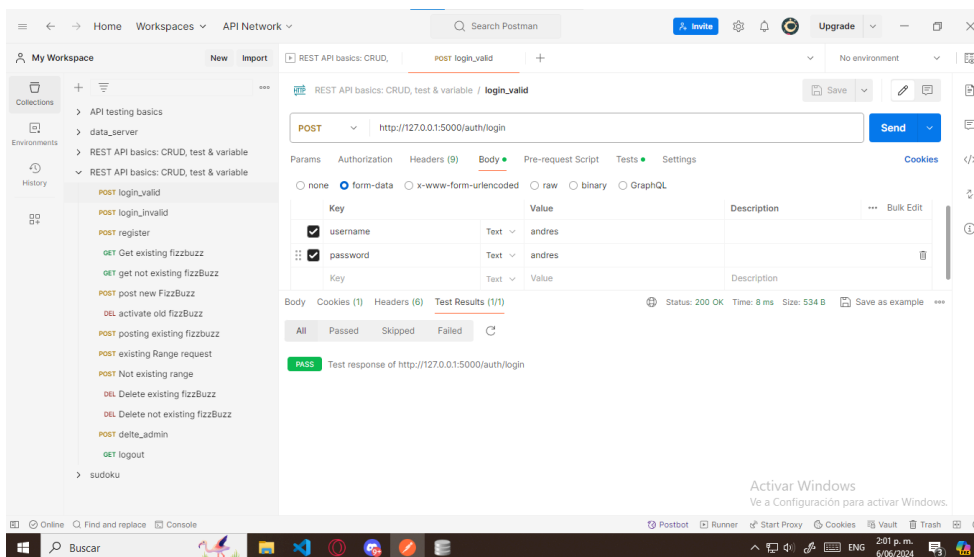
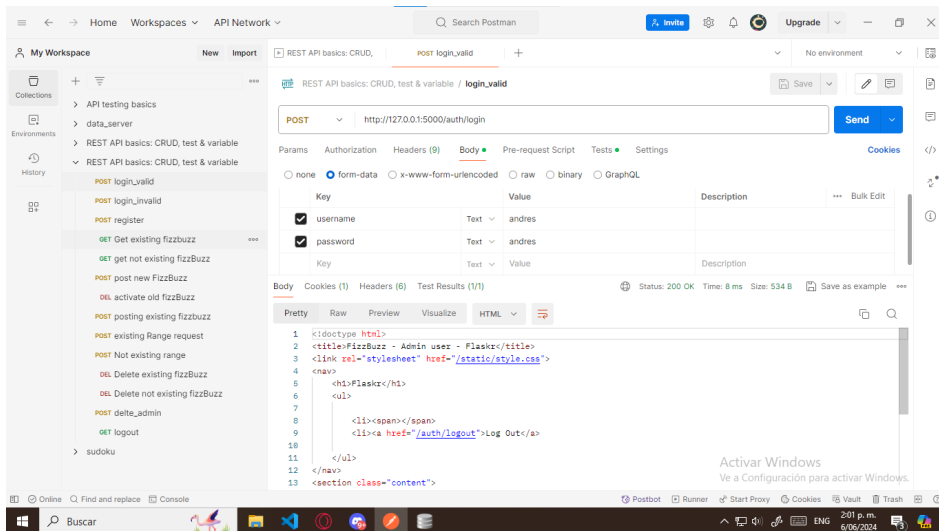
Esta prueba se loguea como administrador y se realiza un delete, la prueba falla ya que aún no se implementan el login.

GREEN

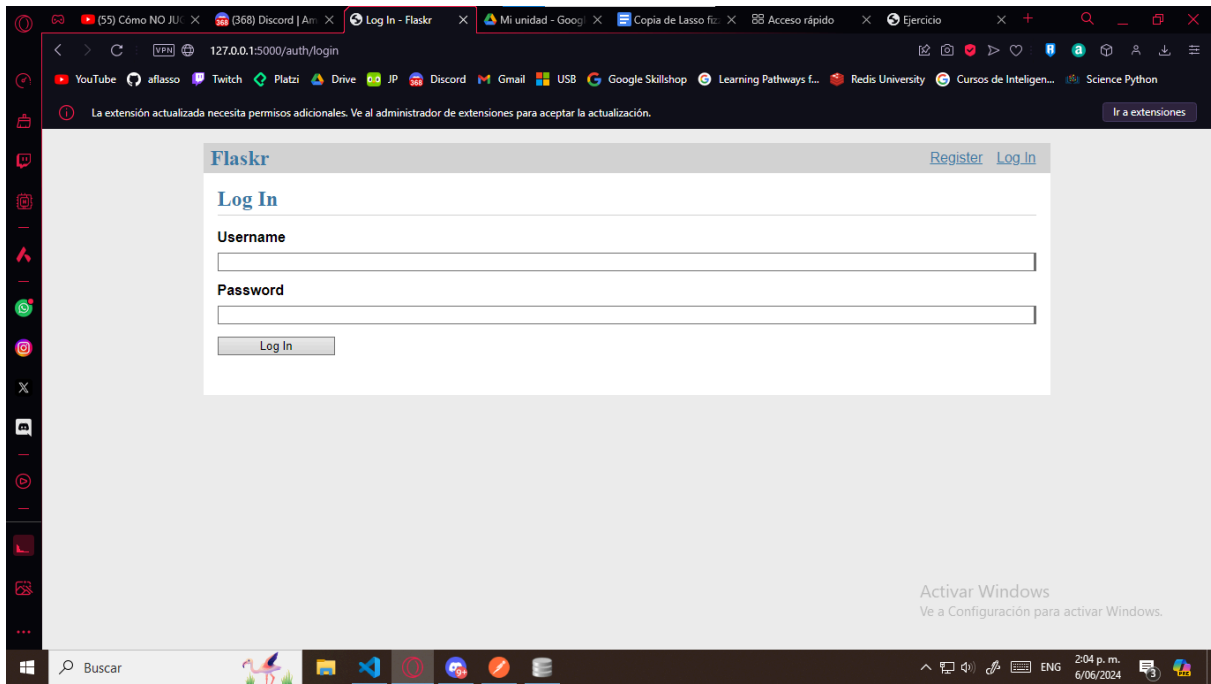
En este punto ya el código pasa las pruebas. Se procederá a explicar cada prueba.



Aquí se hacen solicitudes sin antes realizar una solicitud de login. Como no existe una sesión activa no se puede acceder a la información. Por lo tanto las pruebas realizadas en la iteración anterior fallaran. Lo que confirma que el login funciona y las rutas están protegidas.



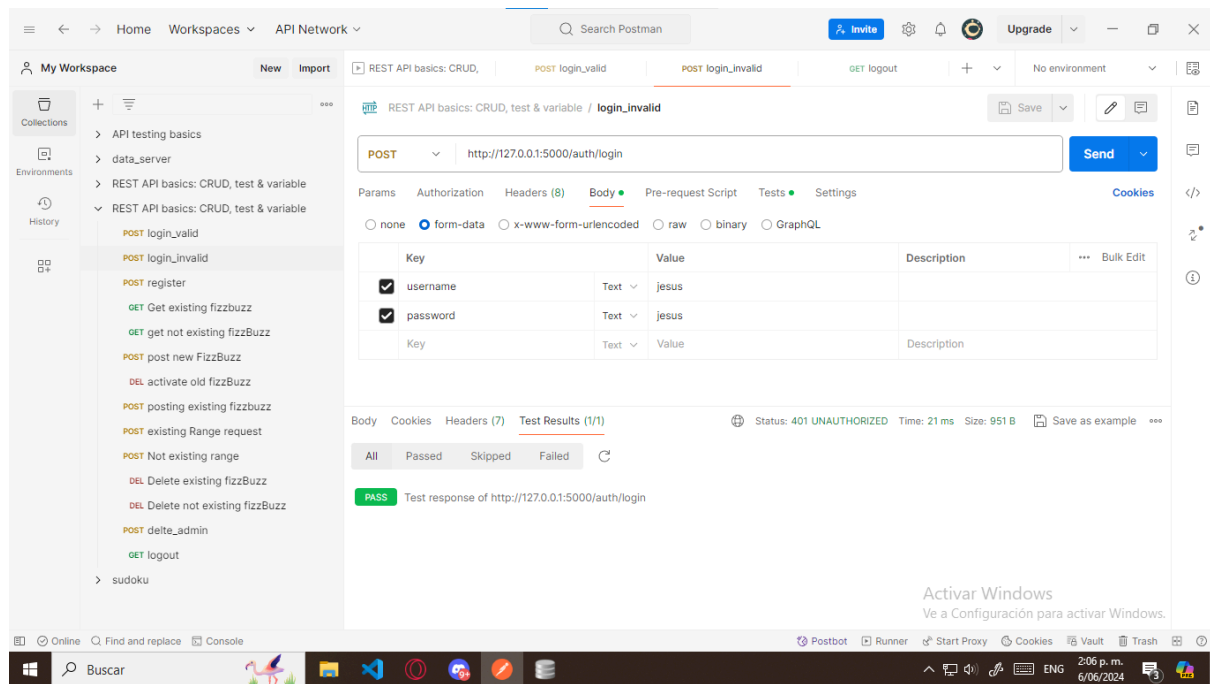
Esta es la prueba individual de un login como administrador. La prueba logra pasar ya que se retorna un estado de 200 y además se puede observar que se crea una cookie de sesión, confirmando que hay una sesión activa.



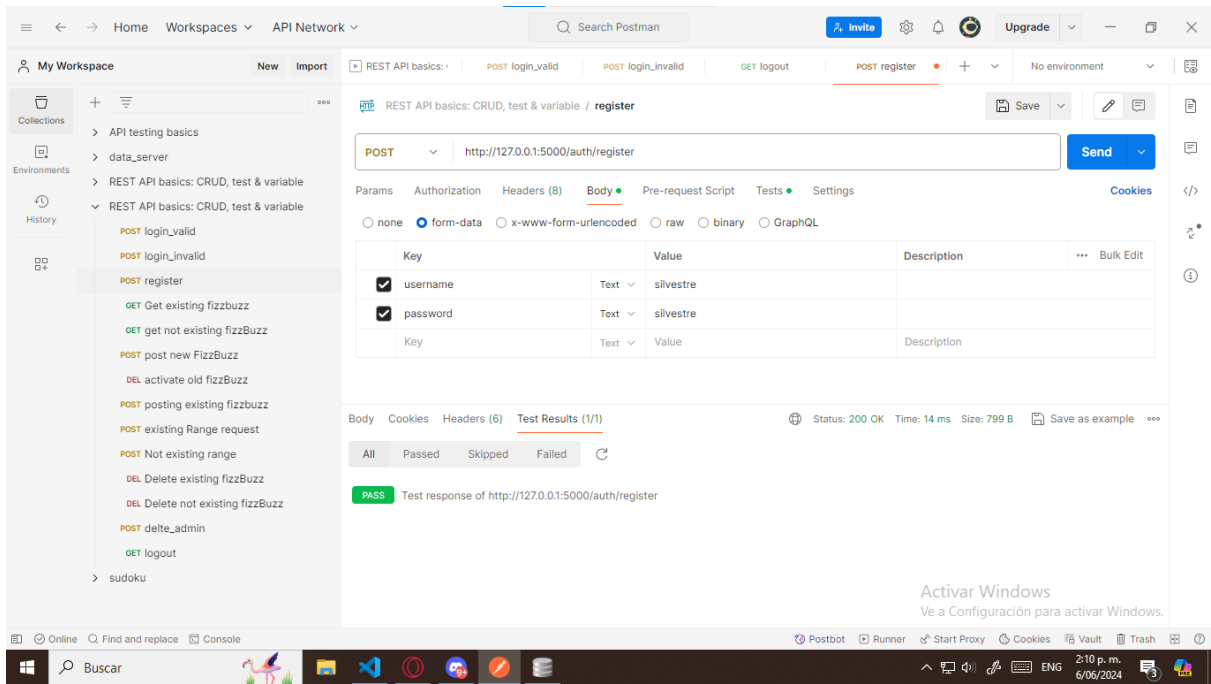
además como se inició con un usuario admin, se le confirma en la pantalla



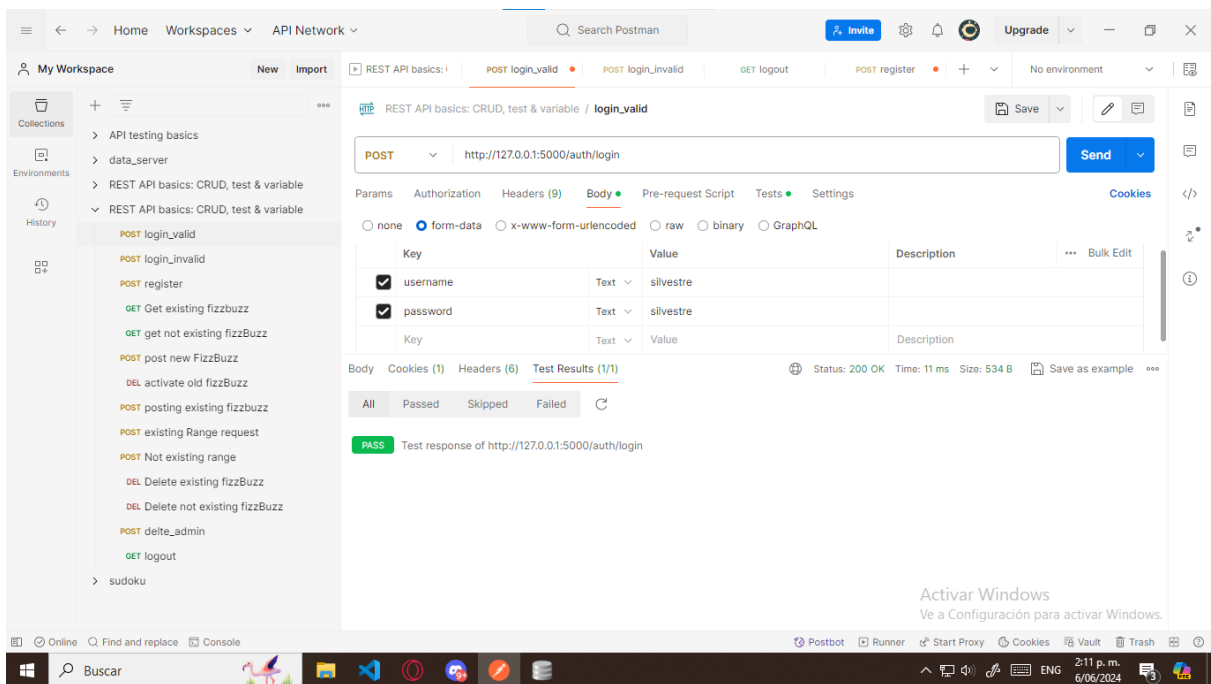
En caso de que inicie sesión un usuario normal, se le confirma en pantalla también

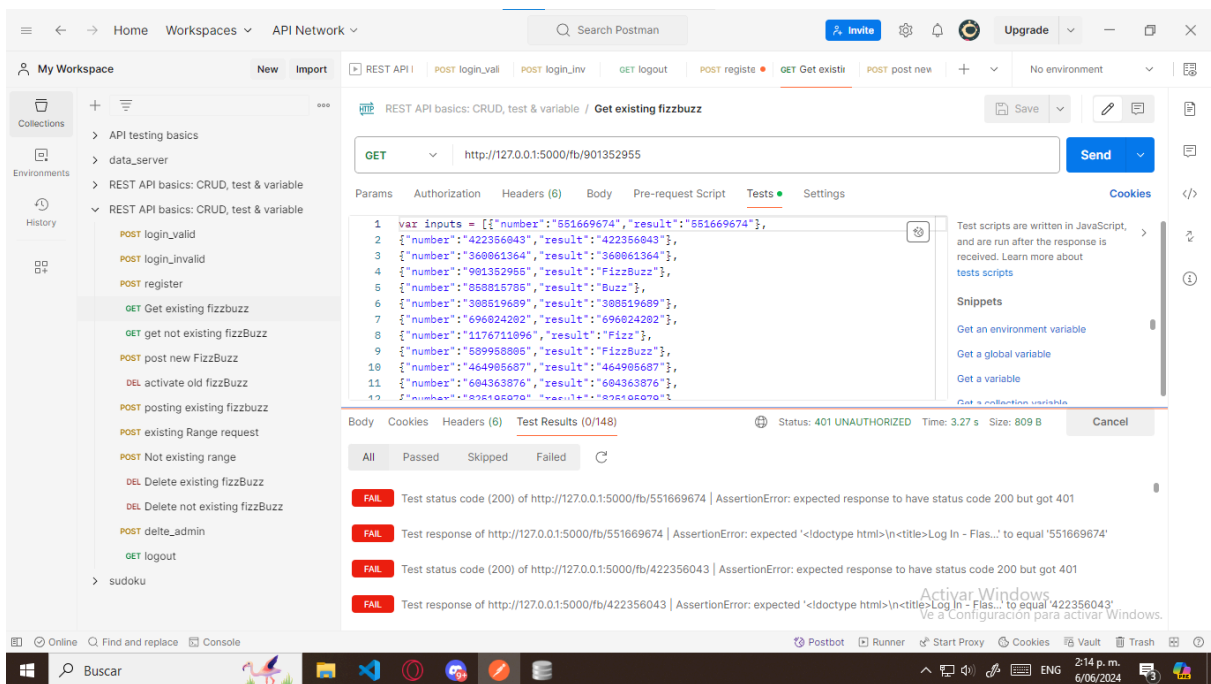
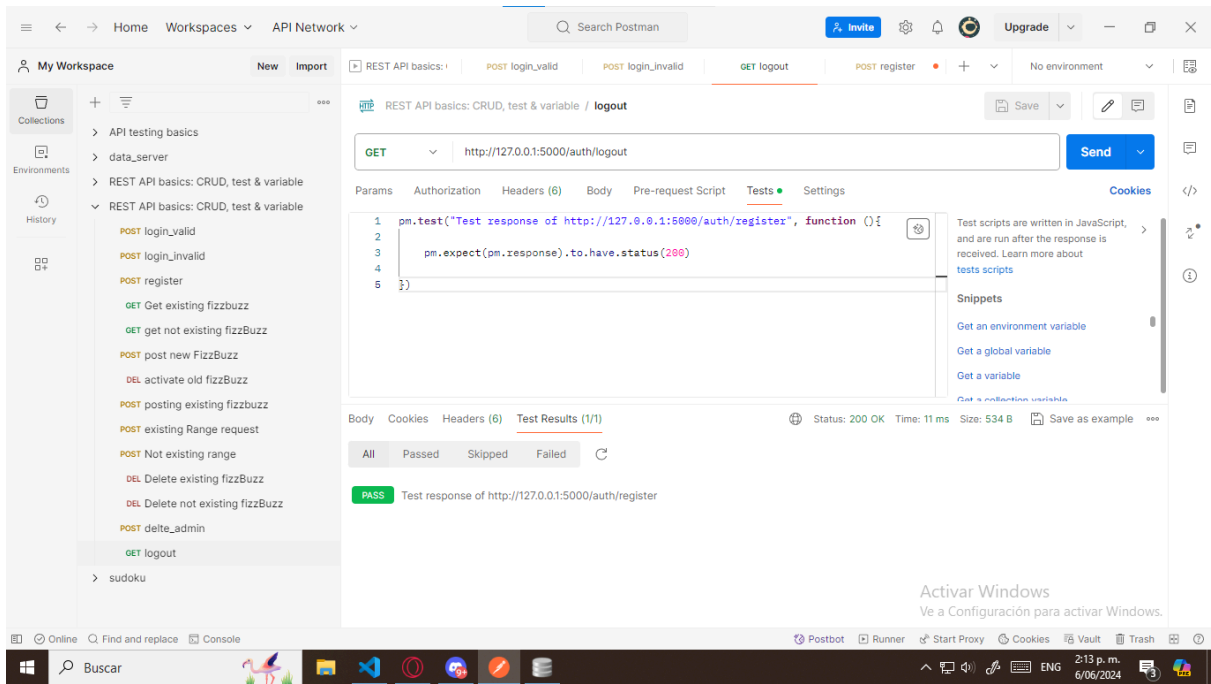


Esta es la prueba para un login invalido. Si se intenta iniciar sesión con un usuario no existente o una contraseña errónea, se retornará un 401. Y no podrá acceder a las rutas protegidas. Ya que no se crea la cookie de sesión

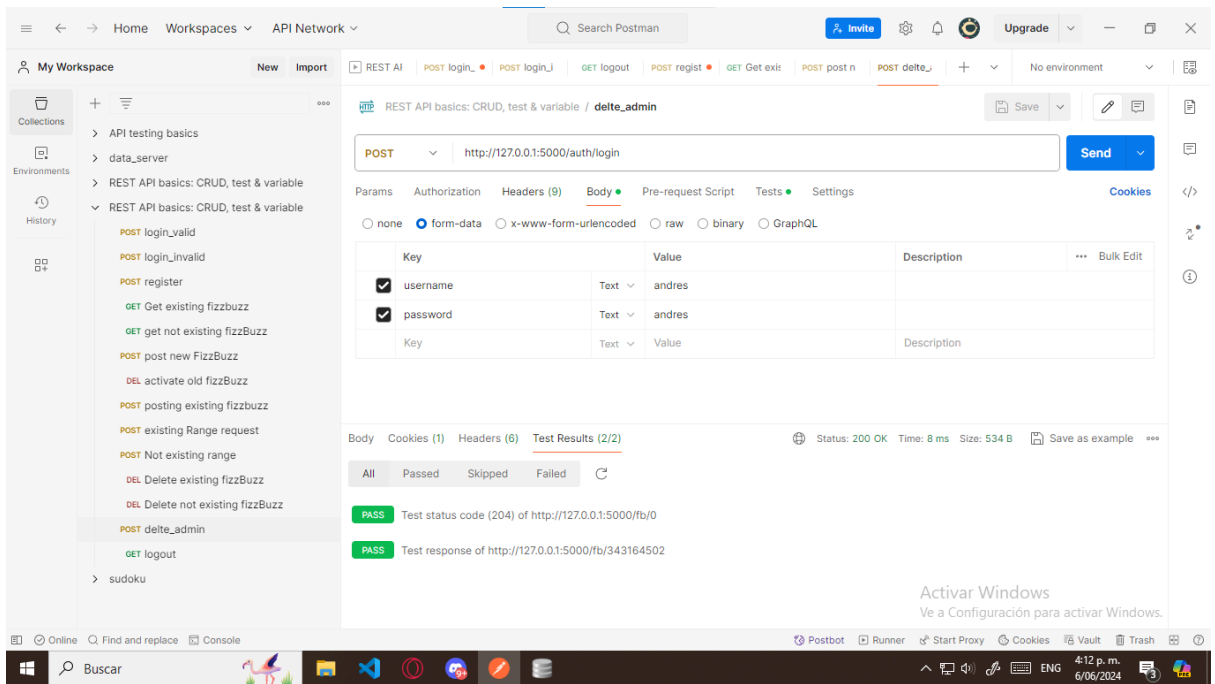
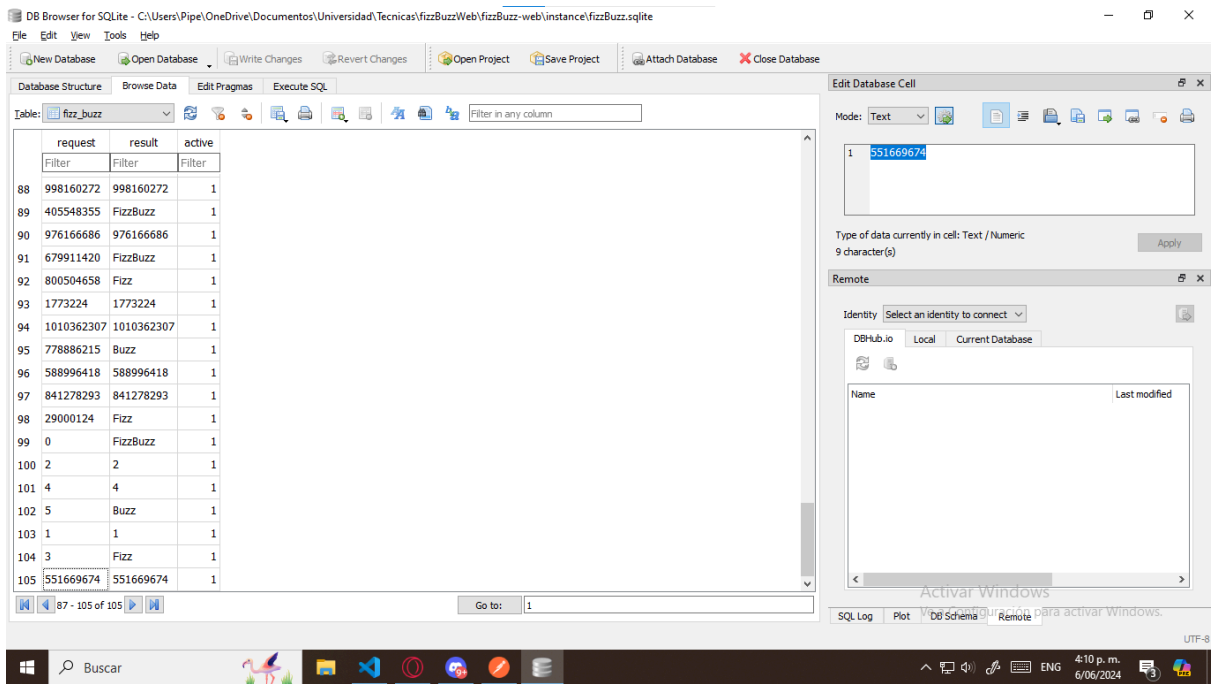


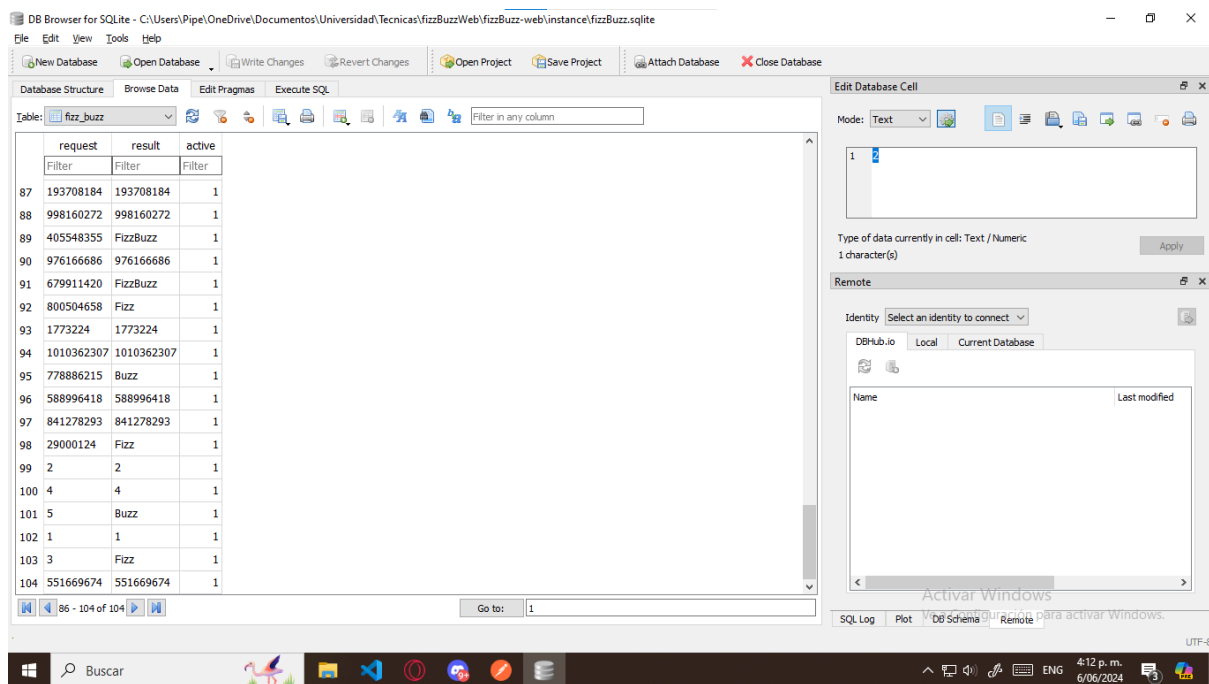
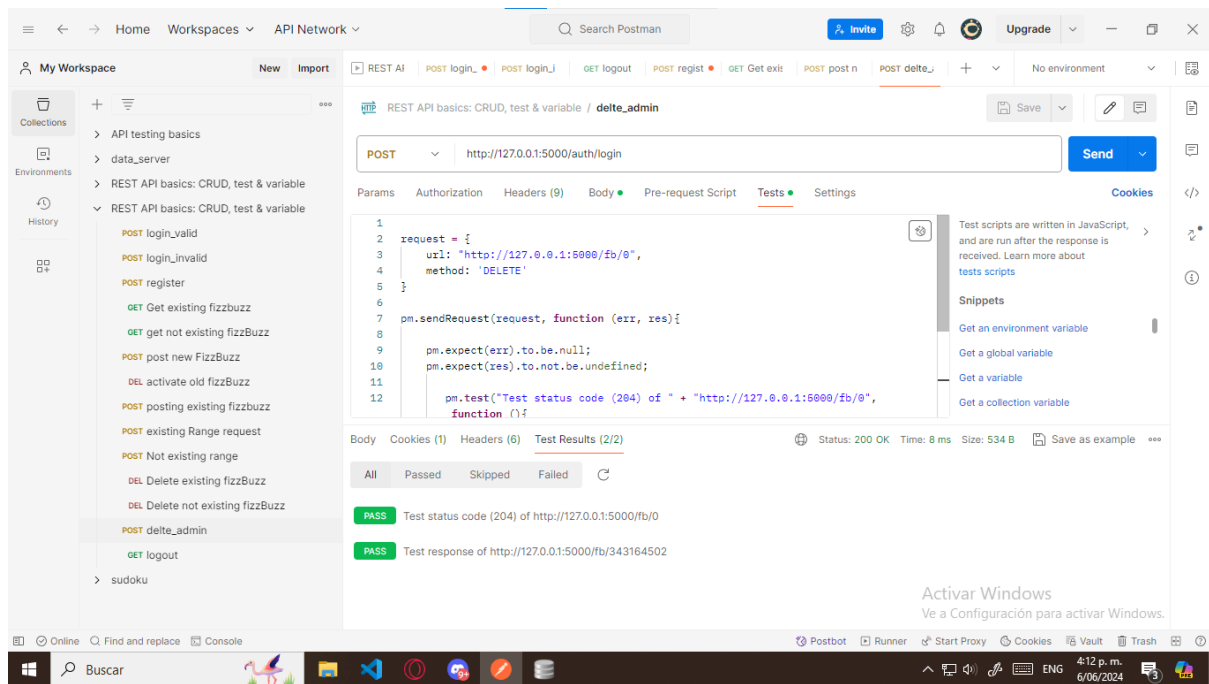
Esta es la prueba de un registro al yo registrar un usuario debe retornar el código 200, además puedo iniciar sesion con ese usuario registrado y también debe retornar un código 200.





Esta es la prueba de un logout cuando realizo un log out me debe retornar un código 200 y si intento acceder a una url protegida después de un logout la url no está disponible.

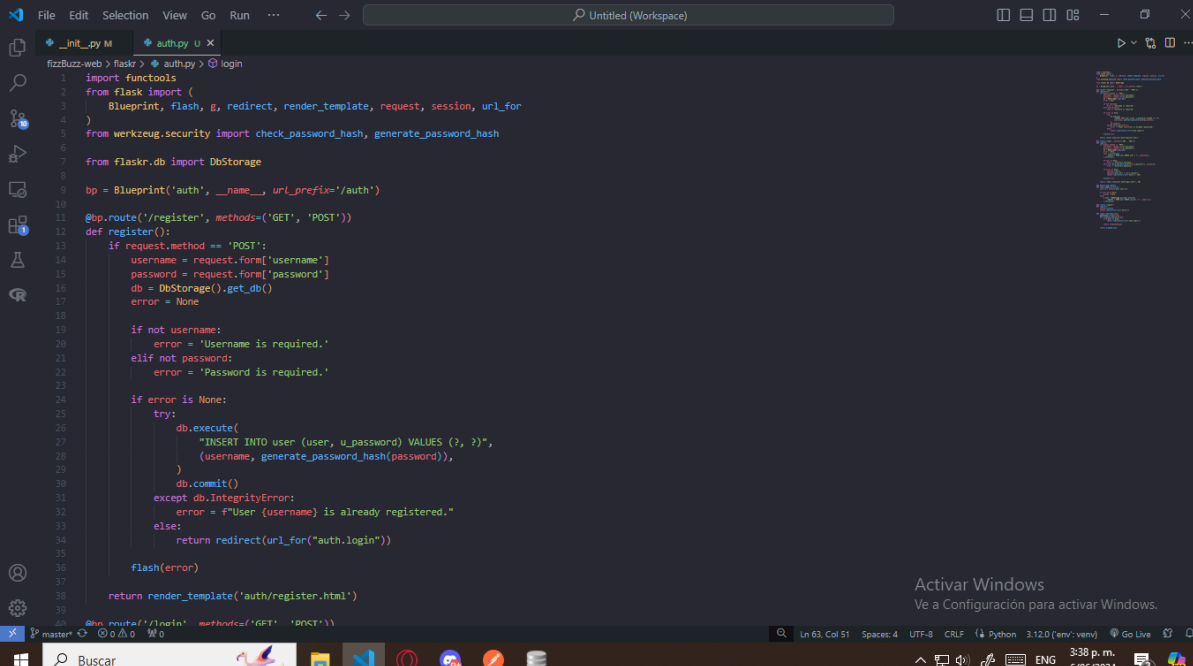




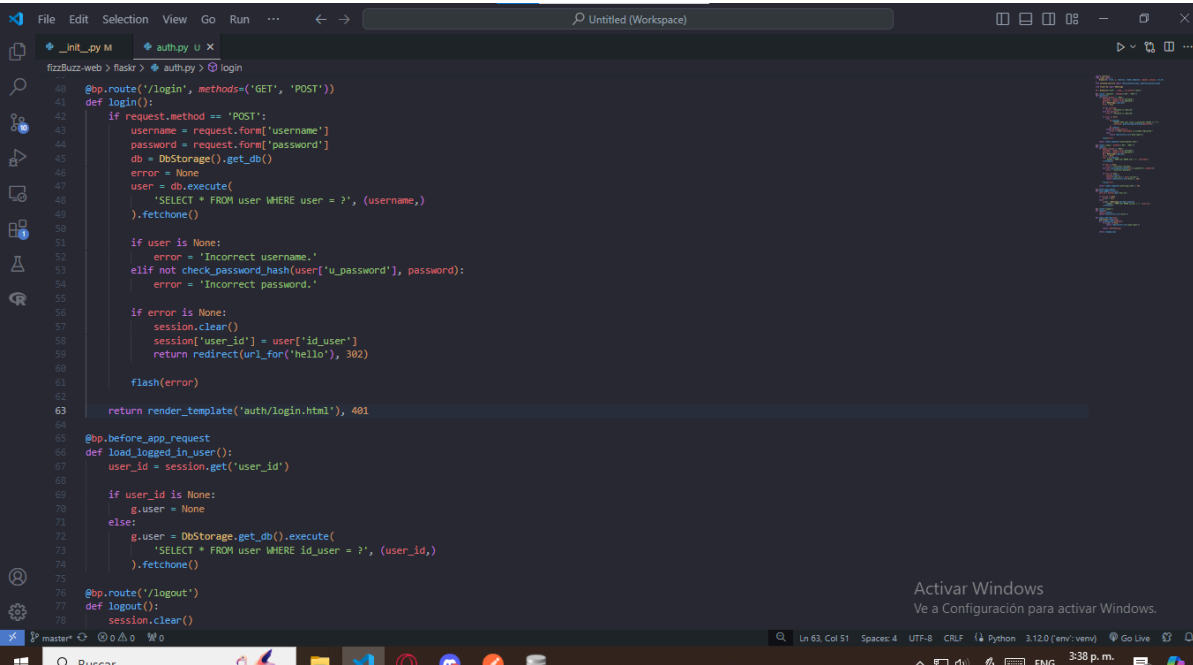
En esta prueba primero se logeo con un usuario admin luego se realizó un delete al número 0. Se pasó la prueba ya que se retornó el código 204 y “Not Content” y en vez de desactivar el numero como ocurría en la iteración anterior el número es directamente borrado de la base de datos

Explicación código

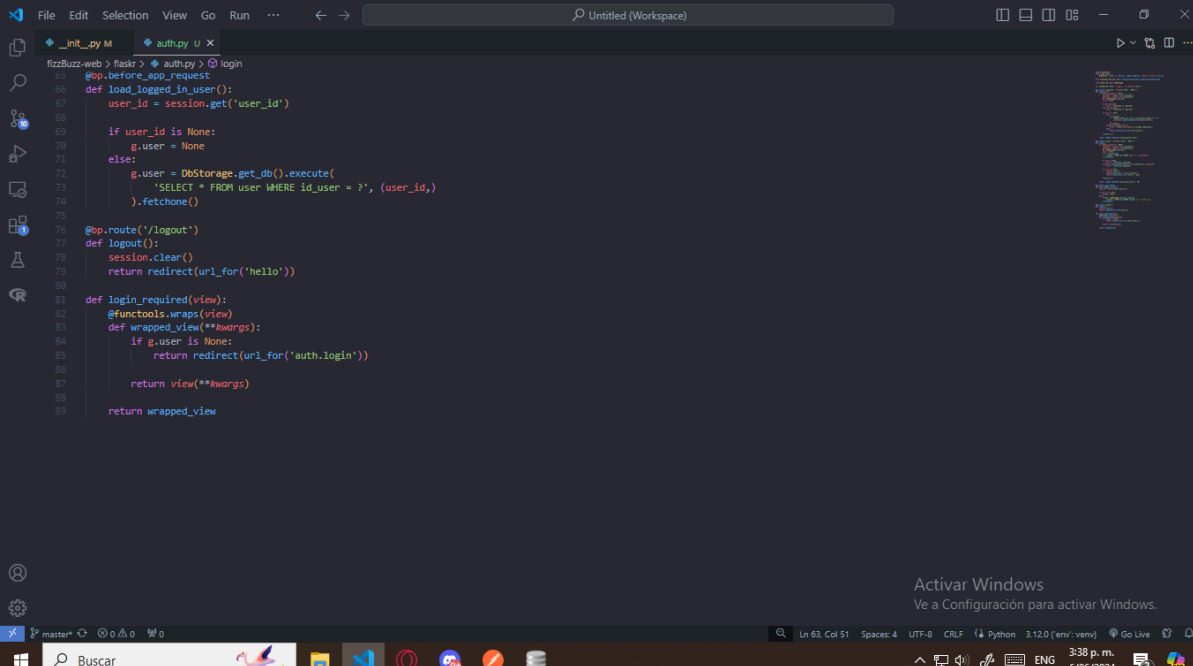
El código es el mismo de la iteración pasada. A este se le añadió el módulo auth que contiene toda la lógica de autenticación.



```
1 import functools
2 from flask import (
3     Blueprint, flash, g, redirect, render_template, request, session, url_for
4 )
5 from werkzeug.security import check_password_hash, generate_password_hash
6
7 from flaskr.db import DbStorage
8
9 bp = Blueprint('auth', __name__, url_prefix='/auth')
10
11 @bp.route('/register', methods=('GET', 'POST'))
12 def register():
13     if request.method == 'POST':
14         username = request.form['username']
15         password = request.form['password']
16         db = DbStorage().get_db()
17         error = None
18
19         if not username:
20             error = 'Username is required.'
21         elif not password:
22             error = 'Password is required.'
23
24         if error is None:
25             try:
26                 db.execute(
27                     "INSERT INTO user (user, u_password) VALUES (?, ?)",
28                     (username, generate_password_hash(password)),
29                 )
30                 db.commit()
31             except db.IntegrityError:
32                 error = 'User (username) is already registered.'
33             else:
34                 return redirect(url_for("auth.login"))
35
36     flash(error)
37
38     return render_template('auth/register.html')
```



```
40 @bp.route('/login', methods=('GET', 'POST'))
41 def login():
42     if request.method == 'POST':
43         username = request.form['username']
44         password = request.form['password']
45         db = DbStorage().get_db()
46         error = None
47         user = db.execute(
48             'SELECT * FROM user WHERE user = ?', (username,)
49         ).fetchone()
50
51         if user is None:
52             error = 'Incorrect username.'
53         elif not check_password_hash(user['u_password'], password):
54             error = 'Incorrect password.'
55
56         if error is None:
57             session.clear()
58             session['user_id'] = user['id_user']
59             return redirect(url_for('hello'), 302)
60
61     flash(error)
62
63     return render_template('auth/login.html'), 401
64
65 @bp.before_app_request
66 def load_logged_in_user():
67     user_id = session.get('user_id')
68
69     if user_id is None:
70         g.user = None
71     else:
72         g.user = DbStorage.get_db().execute(
73             'SELECT * FROM user WHERE id_user = ?', (user_id,)
74         ).fetchone()
75
76 @bp.route('/logout')
77 def logout():
78     session.clear()
```



```
fizzbuzz-web > flask> auth.py login
66 @bp.before_request
67 def load_logged_in_user():
68     user_id = session.get('user_id')
69
70     if user_id is None:
71         g.user = None
72     else:
73         g.user = DbStorage.get_db().execute(
74             'SELECT * FROM user WHERE id_user = ?', (user_id,))
75         .fetchone()
76
77 @bp.route('/logout')
78 def logout():
79     session.clear()
80     return redirect(url_for('hello'))
81
82 def login_required(view):
83     @functools.wraps(view)
84     def wrapped_view(**kwargs):
85         if g.user is None:
86             return redirect(url_for('auth.login'))
87         return view(**kwargs)
88     return wrapped_view
```

Este es el código presente en el tutorial de Flask, así es como flask maneja los login, sign ups y logout. Se crean las rutas que manejan la autenticación, la ruta login, register y logout.

@bp.route('/register', methods=('GET', 'POST')):

Esta view maneja el formulario de registro de un usuario nuevo. Si la solicitud es GET, se renderiza el formulario. Si la solicitud es POST, osea cuando el formulario fue enviado, se abstrae el usuario y la contraseña del formulario y se ingresa en la base de datos. Este registro siempre registra un usuario normal. Los usuarios admin no se pueden crear.

@bp.route('/login', methods=('GET', 'POST')):

Esta view es muy similar a la de register. La diferencia es que esta busca en la base de datos el usuario con la contraseña ingresada y si los datos son válidos se agrega el id del usuario a las cookies de la sesión. En conclusión cuando se realiza un login de un usuario y contraseña válidos se busca ese usuario en la base de datos y se carga su atributo id_usuario en la sesión actual del navegador.

@bp.route('/logout'):

Esta view maneja el logout de una sesión. Cuando es llamada esta view limpia la sesión del navegador actual. Si existe algún id de usuario almacenado, osea logueado, esta view lo elimina y por lo tanto ese usuario ya no está logueado ni tiene los permisos correspondientes.

@bp.before_app_request

def load_logged_in_user():

Esta es una función llamada cada vez que se realiza una request, exactamente antes de ejecutar la view. Lo que hace es abstraer el id de usuario de la sesión activa en el navegador, la cookie. Si está vacía significa que no hay un usuario logueado actualmente, si no está vacía se abstrae ese usuario de la base de datos y se guarda su información en forma de directorio en un lugar llamado "g" el cual utiliza flask para almacenar recursos.

def login_required(view):

Este es un decorador que se les va asignar a las views que necesiten un usuario logueado. Este decorador antes de llamar a la view primero verifica si existe un usuario en el almacenamiento "g", mencionado en la función anterior. Si existe se despliega la view solicitada, si no se redirecciona a la view de login para que realice el inicio de sesión.

```

@app.route('/fb/<num>', methods=('GET', 'POST', 'DELETE'))
@login_required
def get_fizz_buzz(num):

    if request.method == "GET":

        sql_result = system.get_number(num)

        return sql_result

    if request.method == "POST":
        sql_result = system.post_number(num)

        return sql_result

    if request.method == 'DELETE':

        if g.user["u_role"] == 1:
            print("soy admin")
            sql_result = system.delete_hard_number(num)
            return sql_result

        sql_result = system.delete_number(num)

        return sql_result

    return "Not Found", 404

```

```

@app.route('/range', methods=('POST',))
@login_required
def range_fizz_buzz():

    request_body = request.json

    min_value = request_body['min']
    max_value = request_body['max']

    if request.method == "POST":
        print("entre")
        sql_result = system.get_range(min_value, max_value)

    return sql_result

```

Estas son las views que tiene el decorador, por lo tanto para poder acceder a ellas se necesita de un usuario activo en la sesión del navegador.

Delete administrador:

Lo que se hizo para implementar esta funcionalidad fue agregar una condición al método DELETE en la iteración anterior. Si el usuario activo en la sesión es un administrador realiza un delete hard, si no lo es, significa que es un usuario normal y se realiza un delete soft

```
@app.route('/fb/<num>', methods=('GET', 'POST', 'DELETE'))
@login_required
def get_fizz_buzz(num):

    if request.method == "GET":

        sql_result = system.get_number(num)

        return sql_result

    if request.method == "POST":
        sql_result = system.post_number(num)

        return sql_result

    if request.method == 'DELETE':

        if g.user["u_role"] == 1:
            print("im admin")
            sql_result = system.delete_hard_number(num)
            return sql_result

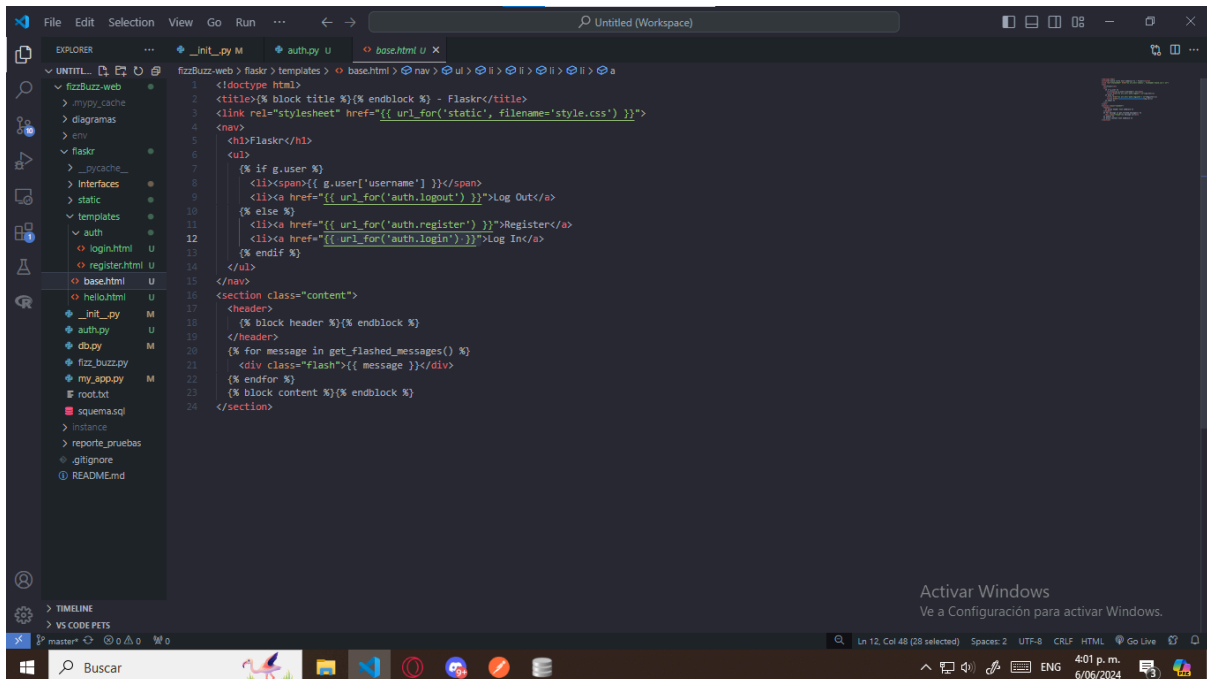
        sql_result = system.delete_number(num)

        return sql_result

    return "Not Found", 404
```


Templates

base.html

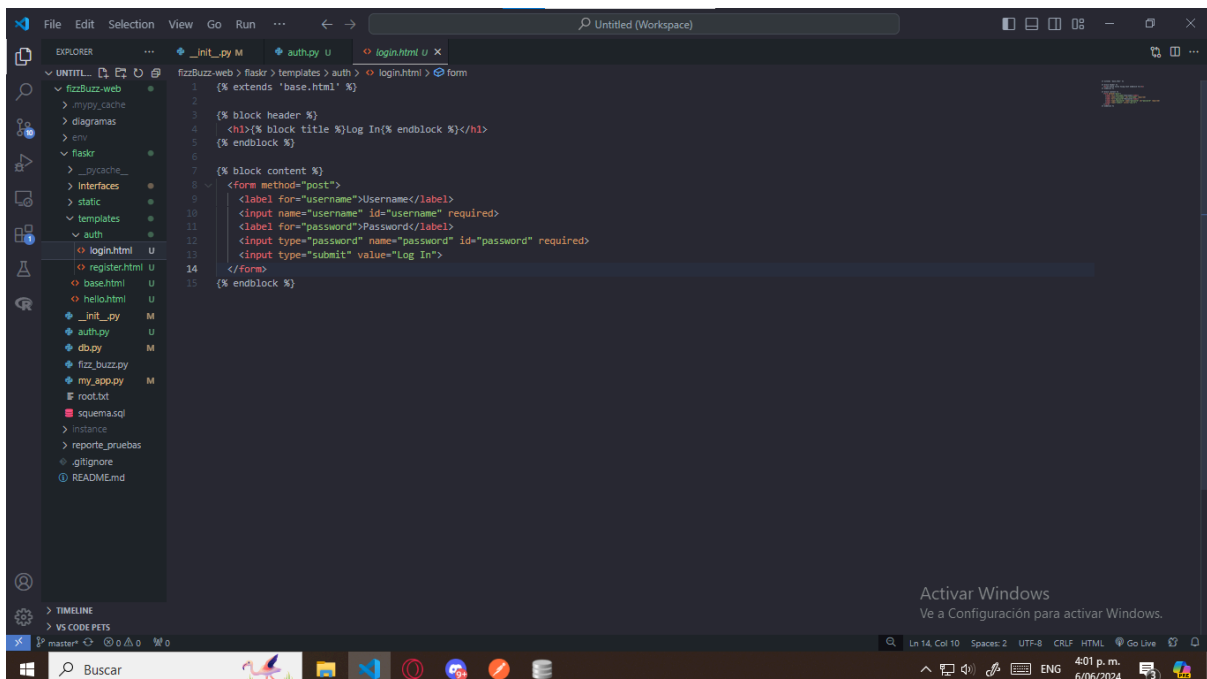


The screenshot shows the Visual Studio Code editor with the 'base.html' file open. The Explorer sidebar on the left shows a project structure for 'fizzbuzz-web' with folders like 'fizzbuzz-web', 'myapp_cache', 'diagrams', 'env', 'flask', and 'templates'. The 'base.html' file is selected in the Explorer. The main editor area displays the following HTML code:

```
1 <!doctype html>
2 <title>{% block title %}{% endblock %} - Flask</title>
3 <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
4 <nav>
5   <h1>Flask</h1>
6   <ul>
7     {% if g.user %}
8     <li><span>{{ g.user['username'] }}</span>
9     <li><a href="{{ url_for('auth.logout') }}">Log Out</a>
10    {% else %}
11    <li><a href="{{ url_for('auth.register') }}">Register</a>
12    <li><a href="{{ url_for('auth.login') }}">Log In</a>
13    {% endif %}
14  </ul>
15 </nav>
16 <section class="content">
17   <header>
18     {% block header %}{% endblock %}
19   </header>
20   {% for message in get_flashed_messages() %}
21     <div class="flash">{{ message }}</div>
22   {% endfor %}
23   {% block content %}{% endblock %}
24 </section>
```

The status bar at the bottom indicates 'Ln 12, Col 48 (28 selected)', 'Spaces: 2', 'UTF-8', 'CRLF', 'HTML', and 'Go Live'. The system tray shows the time as 4:01 p.m. on 6/06/2024.

login.html

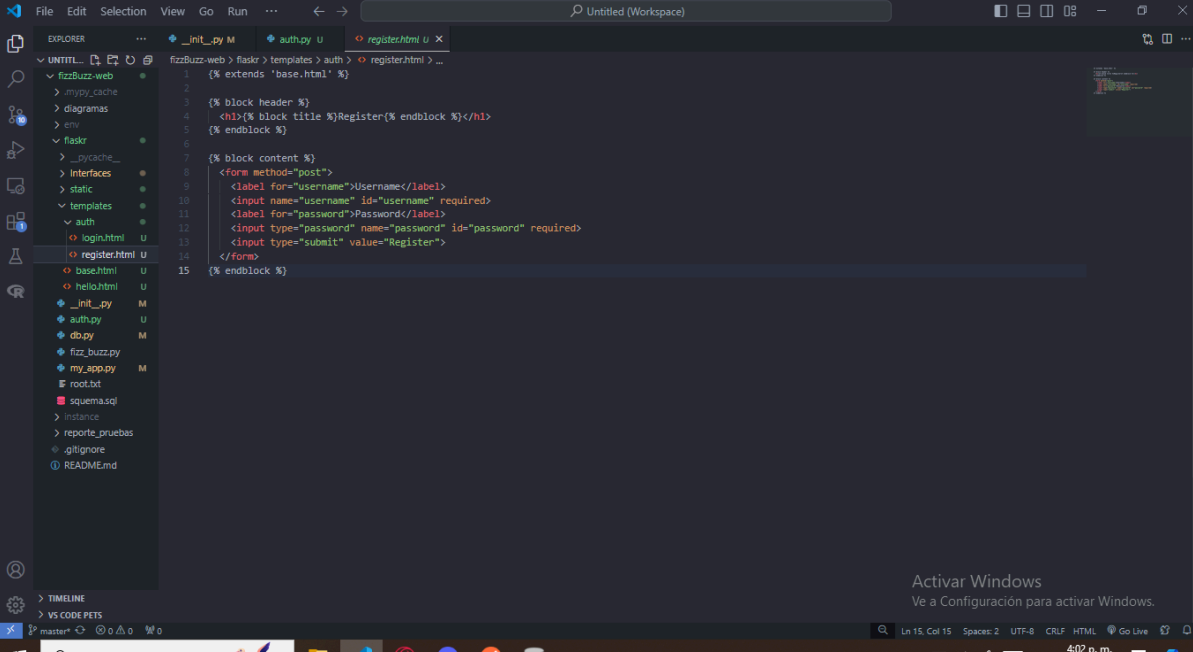


The screenshot shows the Visual Studio Code editor with the 'login.html' file open. The Explorer sidebar on the left shows the same project structure as the previous image. The 'login.html' file is selected in the Explorer. The main editor area displays the following HTML code:

```
1 {% extends 'base.html' %}
2
3 {% block header %}
4   <h1>{% block title %}Log In{% endblock %}</h1>
5 {% endblock %}
6
7 {% block content %}
8   <form method="post">
9     <label for="username">Username</label>
10    <input name="username" id="username" required>
11    <label for="password">Password</label>
12    <input type="password" name="password" id="password" required>
13    <input type="submit" value="Log In">
14  </form>
15 {% endblock %}
```

The status bar at the bottom indicates 'Ln 14, Col 10', 'Spaces: 2', 'UTF-8', 'CRLF', 'HTML', and 'Go Live'. The system tray shows the time as 4:01 p.m. on 6/06/2024.

register.html



The screenshot shows a Visual Studio Code editor window with a workspace named "Untitled (Workspace)". The Explorer sidebar on the left displays a file tree for a project named "fizzBuzz-web". The tree includes folders like ".fizzBuzz-web", ".myapp_cache", "diagrams", "env", "flaskr", and "templates". The "templates" folder is expanded, showing files like "login.html", "register.html", "base.html", and "hello.html". The "register.html" file is selected and its content is displayed in the main editor area. The code is a Jinja2 template that extends "base.html" and contains a registration form. The status bar at the bottom indicates the current position is Line 15, Column 15, with 2 spaces, in UTF-8 encoding, CRLF line endings, and HTML format. The Windows taskbar at the very bottom shows the search bar, taskbar icons, and system clock.

```
1 {% extends 'base.html' %}
2
3 {% block header %}
4 <h1>{% block title %}Register{% endblock %}</h1>
5 {% endblock %}
6
7 {% block content %}
8 <form method="post">
9   <label for="username">Username</label>
10   <input name="username" id="username" required>
11   <label for="password">Password</label>
12   <input type="password" name="password" id="password" required>
13   <input type="submit" value="Register">
14 </form>
15 {% endblock %}
```

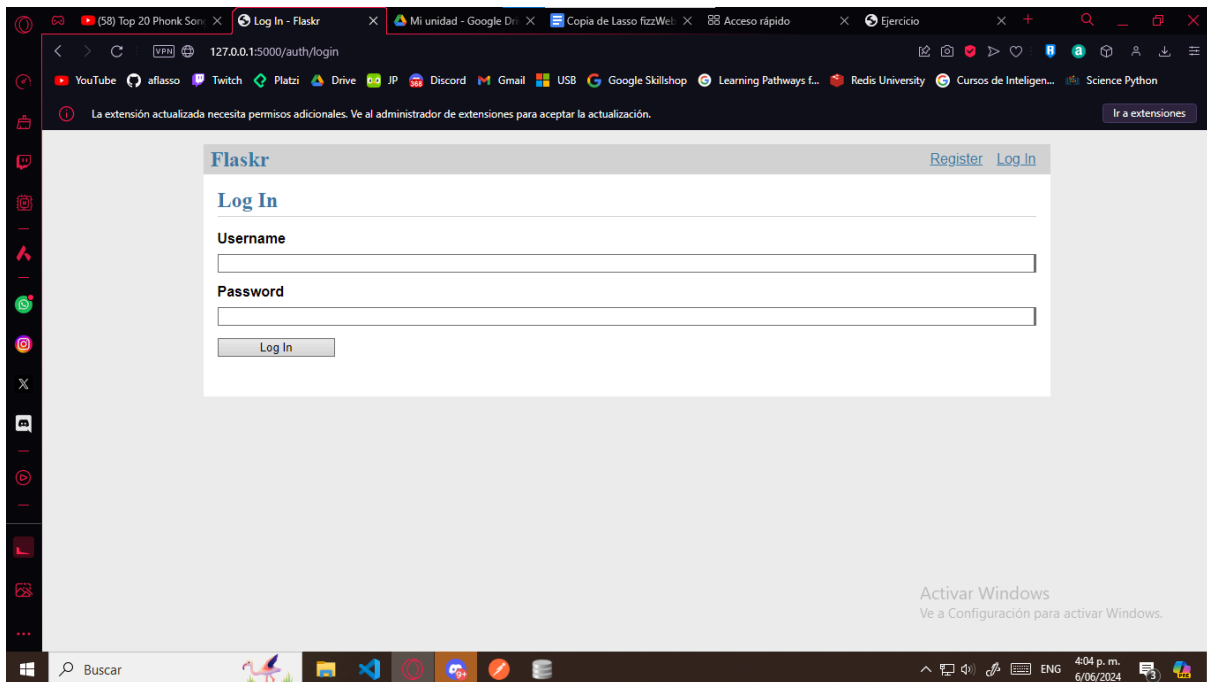
Activar Windows
Ve a Configuración para activar Windows.

Ln 15, Col 15 Spaces: 2 UTF-8 CRLF HTML Go Live

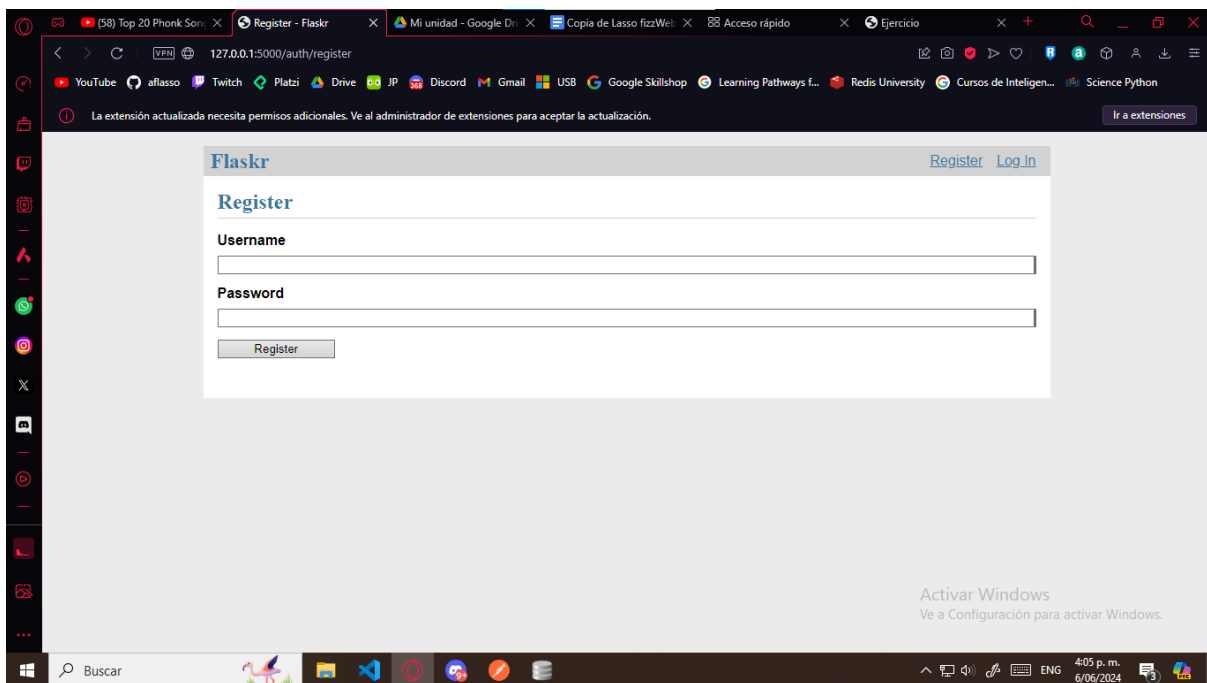
Buscar

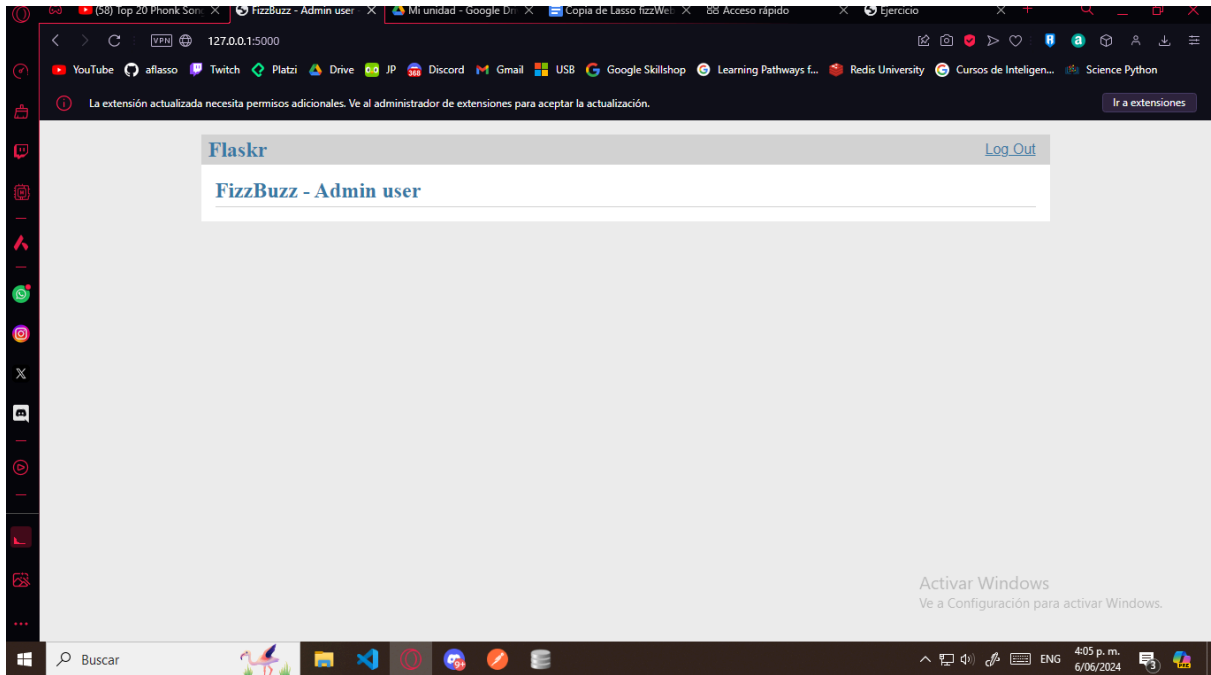
4:02 p.m. 6/06/2024

login.html



register.html

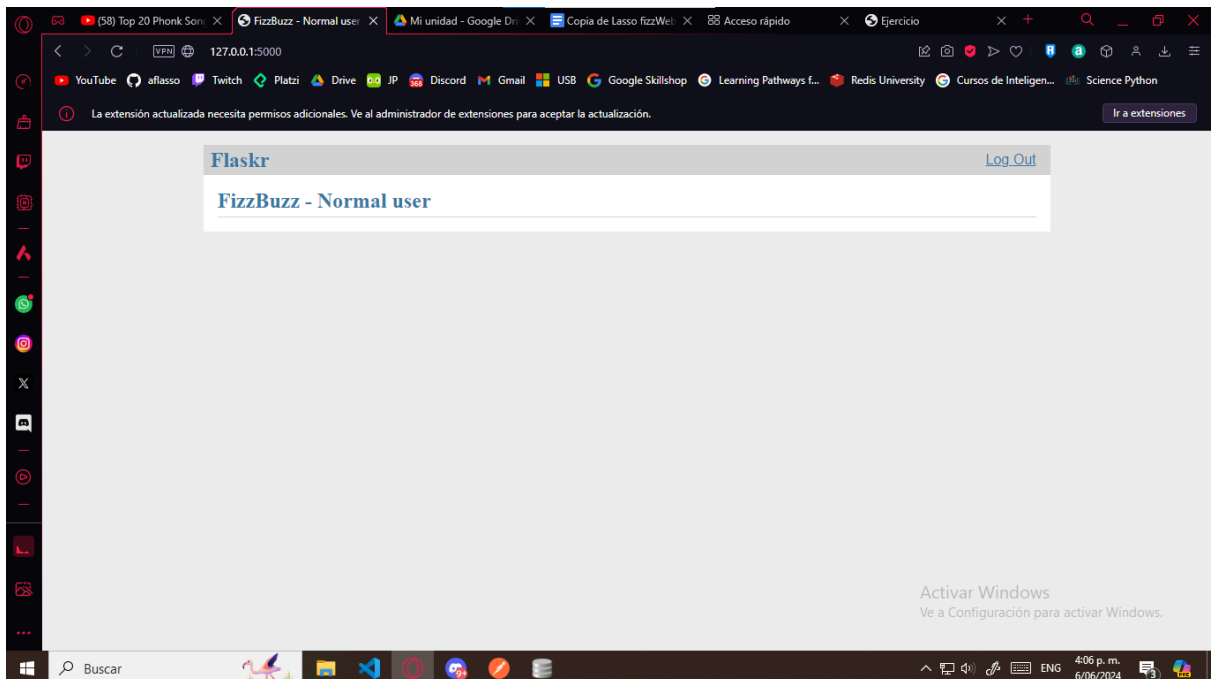




usuario admin

usuario: andres

contraseña: andres



usuario normal

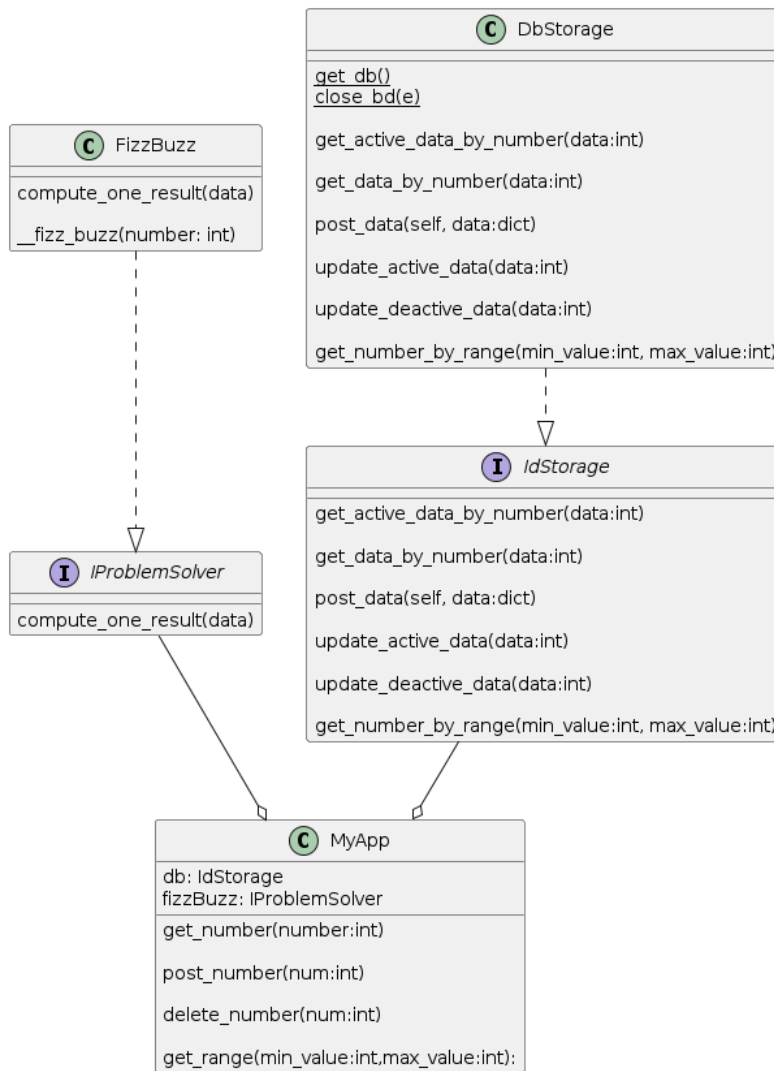
Cualquiera que se ingrese a través de register

Las solicitudes se deben hacer mediante postman o alguna otra herramienta que permite especificar el argumento de la solicitud, GET, POST o DELETE.

Diagramas

Como en esta iteración no se utilizaron clases para realizar la autenticación ni para el delete del administrador no existe un diagrama de clases o de secuencia. Se agregara el diagrama de la anterior iteración.

Diagrama de clases



metacognición

En este proyecto aprendí cómo funciona una autenticación básica y lo importante que es. Realmente no experimenta alguna dificultad ya que en la documentación de flask estaba todo lo necesario, por lo que no fue necesario utilizar alguna inteligencia artificial.