Programación 2

Modelo de datos en la API Blueprints

¿Qué es MVC?

La arquitectura MVC (modelo, vista, controlador) consiste en un patrón de diseño de software que se utiliza para separar el desarrollo de una aplicación en tres componentes: **los datos, la metodología y la interfaz gráfica**. La gran ventaja que posee esta técnica de programación es que permite modificar cada uno de ellos sin necesidad de modificar los demás, lo que permite desarrollar aplicaciones modulares y escalables que se puedan actualizar fácilmente y añadir o eliminar nuevos módulos o funcionalidades de forma paquetizada, ya que cada “paquete” utiliza el mismo sistema con sus vistas, modelos y controladores.

¿Por qué utilizar estos patrones de diseño de software?

Porque es un patrón de diseño de software probado y se sabe que funciona. Con MVC la aplicación se puede desarrollar rápidamente, de forma **modular y mantenible**. Separar las funciones de la aplicación en modelos, vistas y controladores hace que la aplicación sea muy ligera.

El uso de MVC tiene beneficios en términos de **claridad del código, reutilización de componentes y facilidad para realizar cambios o actualizaciones en la aplicación**. Esto puede llevar a un desarrollo más **eficiente** y a una mayor **escalabilidad** del software, lo que puede influir positivamente en el rendimiento de la aplicación.

El diseño modular permite a los diseñadores y a los desarrolladores trabajar conjuntamente, así como realizar rápidamente el prototipado. Esta separación también permite hacer cambios en una parte de la aplicación sin que las demás se vean afectadas.

Modelo

El Modelo se encarga de manipular, gestionar y actualizar los **datos**. Si se utiliza una base de datos aquí es donde se realizan las consultas, búsquedas, filtros y actualizaciones.

Vista

La Vista sirve para mostrarle al usuario final la **interfaz gráfica** (pantallas, ventanas, páginas, formularios...) como resultado de una solicitud enviada a través del controlador. Desde la perspectiva del programador este componente es el que se encarga del **frontend**; la programación de la interfaz de usuario si se trata de un aplicación de escritorio, o bien, la visualización de las páginas web (CSS, HTML, HTML5 y Javascript).

Controlador

El Controlador es el **componente principal** de la aplicación, donde se especifican los métodos y funcionalidades que una aplicación (o módulo de una aplicación) tienen que realizar. Se encarga de gestionar las instrucciones que se reciben, atenderlas y procesarlas. A través del controlador se realizan las consultas al modelo (una búsqueda por ejemplo), y una vez se hayan obtenido dichos datos, se envía a la vista las instrucciones necesarias para poder mostrarlos de una forma legible para el usuario.

Rutas

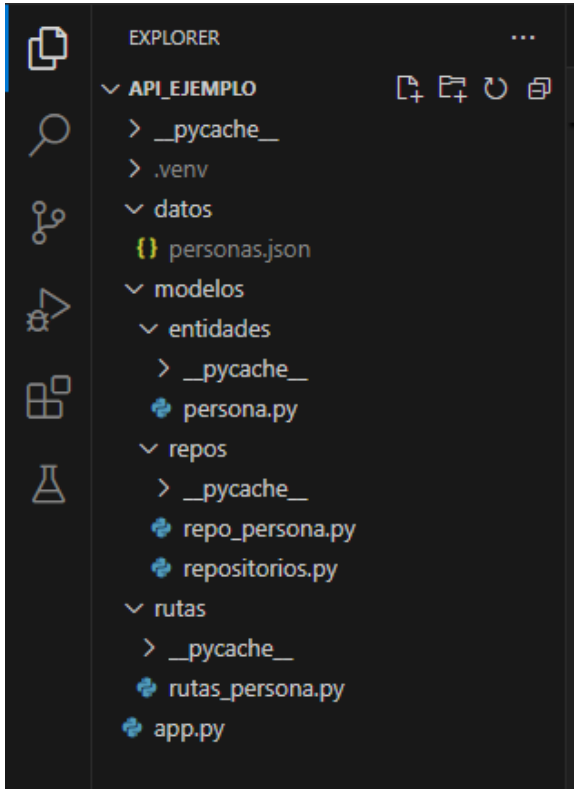
Una ruta es esencialmente la dirección o URL específica que un cliente (otra aplicación, un navegador, etc.) utiliza para solicitar un recurso o realizar una acción en tu aplicación. Podemos imaginar las rutas como las direcciones de las casas en una ciudad: cada dirección te lleva a un lugar específico.

En el patrón MVC, cada ruta está asociada a un controlador. Cuando un cliente envía una solicitud a una determinada ruta, Flask (o cualquier otro framework web) busca el controlador correspondiente y ejecuta la función asociada a esa ruta. Esta función, a su vez, interactuará con el modelo para obtener los datos necesarios y devolver una respuesta al cliente.

Estructura de Archivos propuesta (patrón MVC)

```
mi_proyecto/
├── app.py
├── rutas/
│   ├── rutas_entidad1.py
│   ├── ...
│   └── rutas_entidadN.py
├── modelos/
│   ├── entidades/
│   │   ├── entidad1.py
│   │   ├── ...
│   │   └── entidadN.py
│   └── repos/
│       ├── repo_entidad1.py
│       ├── ...
│       ├── repo_entidadN.py
│       └── repositorios.py
└── datos/
    ├── entidad1.json
    ├── ...
    └── entidadN.json
```

Propuesta de estructura a utilizar



Estructura de Archivos propuesta

```
mi_proyecto/
├── app.py
├── rutas/
│   ├── rutas_entidad1.py
│   ├── ...
│   └── rutas_entidadN.py
├── modelos/
│   ├── entidades/
│   │   ├── entidad1.py
│   │   ├── ...
│   │   └── entidadN.py
│   └── repos/
│       ├── repo_entidad1.py
│       ├── ...
│       ├── repo_entidadN.py
│       └── repositorios.py
└── datos/
    ├── entidad1.json
    ├── ...
    └── entidadN.json
```

- **app.py**: Actúa como punto de entrada de la aplicación y puede encargarse de la configuración inicial, como cargar rutas y configurar las dependencias necesarias (por ejemplo, instanciar los repositorios)
- **rutas/**: (también se podría llamar *controladores*) Los archivos en esta carpeta son controladores que definen las rutas (endpoints) de la API. Se encargan de interactuar con los repositorios y de gestionar las peticiones HTTP.
- **modelos/entidades/** : Define las clases de negocio con la lógica de negocio. Estas clases representan a las entidades de la aplicación y contienen la lógica necesaria para validar, manipular y representar los datos.
- **modelos/repos/** : Las clases repositorio de esta carpeta gestionan el acceso a los datos en los archivos JSON y encapsulan las operaciones de carga, guardado y modificación de datos. ***repositorios.py*** es útil para inicializar y centralizar las instancias de los repositorios, permitiendo que la aplicación use los mismos repositorios compartidos.
- **datos/** : Almacena los archivos JSON con los datos de las entidades

¿Como hacemos para definir rutas fuera de app.py? Blueprints

Un "blueprint" en Flask es una forma de organizar y estructurar una aplicación web en Python.

Los "blueprints" permiten dividir una aplicación en componentes más pequeños y reutilizables, lo que hace que el código sea más modular y mantenible.

Se utilizan comúnmente para agrupar rutas relacionadas, vistas, plantillas y recursos estáticos.

Los "blueprints" son útiles en situaciones en las que tenemos una aplicación web grande o compleja que necesita ser dividida en secciones o módulos independientes. Ej:

- **Aplicaciones grandes**: Si tenemos una aplicación con muchas rutas y vistas, los "blueprints" permiten dividirla en partes más manejables.
- **APIs RESTful**: Para organizar rutas y vistas relacionadas con una API RESTful, puedes crear un "blueprint" para cada recurso o entidad.
- **Extensiones de aplicaciones**: Si se desea agregar funcionalidades a una aplicación existente, los "blueprints" facilitan la incorporación de nuevas características de manera organizada.

¿Cómo lo usamos?

```python
from flask import Blueprint     # Archivo de ej.: rutas_recurso.py

# Crear un objeto Blueprint
bp_recurso = Blueprint('bp_recurso', __name__)

# Define rutas y la lógica relacionadas con este "blueprint"
@bp_recurso .route('/recurso', methods=["GET"])
def operaciones1():
    return 'Este es el resultado de lo solicitado en recurso'

@bp_recurso .route('/recurso/<int:id>', methods=["GET"])
def operaciones2(id):
    return 'Este es el resultado de lo solicitado en recurso/id'
```

En este ejemplo creamos un "blueprint" llamado "my_blueprint" que contiene dos rutas.

Luego, registramos ese "blueprint" en la aplicación principal usando app.register_blueprint(my_blueprint).

Aclaración: no existe una convención estricta para nombrar los "blueprints" en Flask, pero es una buena práctica elegir nombres descriptivos que reflejen la funcionalidad o el propósito del conjunto de rutas agrupadas bajo ese "blueprint".

app.py (ejemplo)

```python
from flask import Flask
from rutas.rutas_recurso import bp_recurso #importo el bp del archivo

app = Flask(__name__)

# Registrar el "blueprint" en la aplicación
app.register_blueprint(bp_recurso)

if __name__ == '__main__':
    app.run()
```

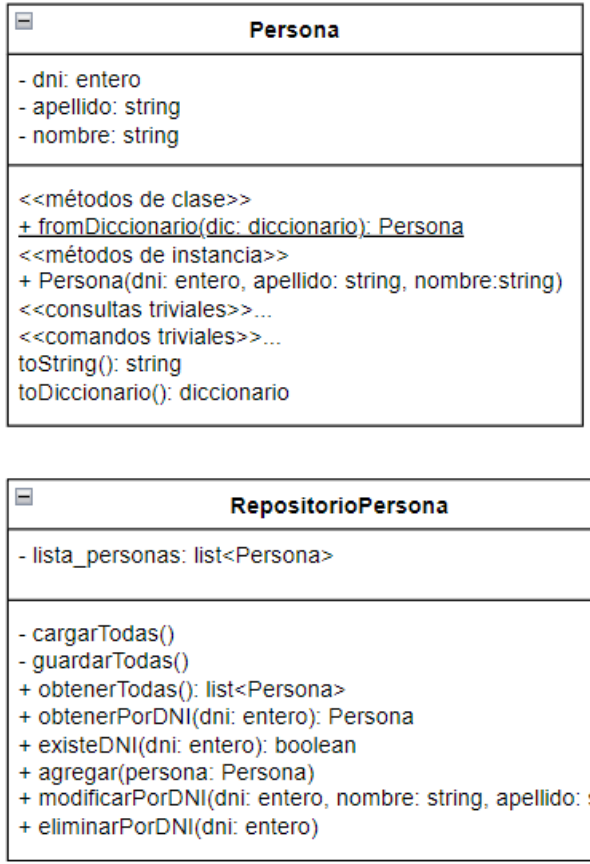
Ahora, cuando se acceda a las rutas */recurso* y */recurso/id*, Flask dirigirá las solicitudes al "blueprint" correspondiente.

Los "blueprints" son especialmente útiles para separar la lógica de diferentes partes de la aplicación, como la autenticación, la gestión de usuarios, la API REST, etc.

Usar "blueprints" ayuda a mantener tu código organizado y facilita la colaboración en proyectos más grandes al separar las funcionalidades en componentes independientes y reutilizables.

Ejemplo de API aplicando todo

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```



```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
from flask import Flask
from rutas.rutas_persona import personas_bp

# instancio la aplicación Flask
app = Flask(__name__)

# registro el blueprint de personas
# el blueprint es una forma de agrupar rutas
app.register_blueprint(personas_bp)

if __name__ == '__main__':
    app.run(debug=True)
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
class Persona:
    @classmethod
    def fromDiccionario(cls, diccionario: dict):
        if not isinstance(diccionario, dict):
            raise ValueError('El parámetro debe ser un diccionario')
        if 'dni' not in diccionario:
            raise ValueError('El diccionario debe tener la clave dni')
        if 'nombre' not in diccionario:
            raise ValueError('El diccionario debe tener la clave nombre')
        if 'apellido' not in diccionario:
            raise ValueError('El diccionario debe tener la clave apellido')
        return cls(diccionario['dni'], diccionario['nombre'], diccionario['apellido'])

    def __init__(self, dni: int, nombre:str, apellido:str):
        if not isinstance(dni, int) or dni < 0:
            raise ValueError('El dni debe ser un entero positivo')
        if not isinstance(nombre, str) or nombre == '' or nombre.isspace():
            raise ValueError('El nombre debe ser un string')
        if not isinstance(apellido, str) or apellido == '' or apellido.isspace():
            raise ValueError('El apellido debe ser un string')
        self.__dni = dni
        self.__nombre = nombre
        self.__apellido = apellido
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
    def obtenerDni(self):
        return self.__dni

    def obtenerNombre(self):
        return self.__nombre

    def obtenerApellido(self):
        return self.__apellido

    def establecerNombre(self, nombre:str):
        if not isinstance(nombre, str) or nombre == '' or nombre.isspace():
            raise ValueError('El nombre debe ser un string')
        self.__nombre = nombre

    def establecerApellido(self, apellido:str):
        if not isinstance(apellido, str) or apellido == '' or apellido.isspace():
            raise ValueError('El apellido debe ser un string')
        self.__apellido = apellido

    def toDiccionario(self):
        return {'dni': self.__dni,'nombre': self.__nombre,'apellido':
self.__apellido}

    def __str__(self):
        return f'{self.__dni} - {self.__nombre} {self.__apellido}'
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
from modelos.entidades.persona import Persona
import json

class RepositorioPersona:
    FILE_PATH = "datos\personas.json"

    def __init__(self):
        """Inicializa el repositorio de personas con los datos del
archivo."""
        self.__personas = self.__cargarTodas()

    def __cargarTodas(self):
        lista =[]
        try:
            with open(RepositorioPersona.FILE_PATH, 'r') as file:
                personas = json.load(file)
                for p in personas:
                    lista.append(Persona.fromDiccionario(p) )

        except FileNotFoundError:
            print("No se encontró el archivo con datos de personas.")

        return lista
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
def __guardarTodas(self):
    try:
        lista = []
        for p in self.__personas:
            lista.append(p.toDiccionario())

        with open(RepositorioPersona.FILE_PATH, 'w') as file:
            json.dump(lista, file, indent=4)
    except FileNotFoundError:
        print("No se encontró el archivo con datos de personas.")

# retornar todas las personas
def obtenerTodas(self):
    return self.__personas


# obtener una persona por dni
def obtenerPorDni(self, dni: int):
    for p in self.__personas:
        if p.obtenerDni() == dni:
            return p
    return None
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
# agregar una persona
def agregar(self, persona: Persona):
    if not isinstance(persona, Persona):
        raise ValueError('El parámetro debe ser una Persona')
    if self.existeDni(persona.obtenerDni()):
        raise ValueError('Ya existe una persona con ese DNI')
    self.__personas.append(persona)
    self.__guardarTodas()

# verificar si existe una persona por dni
def existeDni(self, dni: int)->bool:
    for p in self.__personas:
        if p.obtenerDni() == dni:
            return True
    return False
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
# modificar el nombre de una persona por dni
def modificarPorDni(self, dni: int, nombre: str, apellido:str)->bool:
    for p in self.__personas:
        if p.obtenerDni() == dni:
            p.establecerNombre(nombre)
            p.establecerApellido(apellido)
            self.__guardarTodas()
            return True
    return False

# eliminar una persona por dni
def eliminarPorDni(self, dni: int)->bool:
    for p in self.__personas:
        if p.obtenerDni() == dni:
            self.__personas.remove(p)
            self.__guardarTodas()
            return True
    return False
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

Este patrón es útil para evitar múltiples instancias del repositorio, lo que podría llevar a inconsistencias en los datos si diferentes partes del programa estuvieran trabajando con diferentes instancias. Al mantener una única instancia global, se garantiza que todas las operaciones sobre el repositorio de personas se realicen sobre el mismo conjunto de datos.

```python
from modelos.repos.repositorio_persona import RepositorioPersona

# defino un repositorio para cada entidad que tuviera en el proyecto
# lo declaro como variable global para que sea accesible desde cualquier
parte del proyecto
repo_Personas = None

def obtenerRepositorioPersonas():
    # declaro que voy a usar la variable global
    global repo_Personas
    # instancio el repositorio de personas
    if repo_Personas is None:
        repo_Personas = RepositorioPersona()
    return repo_Personas
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```python
from flask import Blueprint, request, jsonify
from modelos.entidades.persona import Persona
from modelos.repos.repositorios import obtenerRepositorioPersonas

repo_Personas = obtenerRepositorioPersonas()

personas_bp = Blueprint('personas_bp', __name__)

@personas_bp.route('/personas', methods=['GET'])
def obtener_personas():
    return jsonify([p.toDiccionario() for p in repo_Personas.obtenerTodas()])

@personas_bp.route('/personas/<int:dni>', methods=['GET'])
def obtener_persona(dni):
    persona_encontrada = repo_Personas.obtenerPorDni(dni)
    if isinstance(persona_encontrada, Persona):
        return jsonify(persona_encontrada.toDiccionario())
    else:
        return jsonify({'error': 'No se encontró la persona'}), 404

@personas_bp.route('/personas/<int:dni>', methods=['DELETE'])
def eliminar_persona(dni):
    if repo_Personas.eliminarPorDni(dni):
        return jsonify({'mensaje': 'Persona eliminada'}), 200
    else:
        return jsonify({'error': 'No se encontró la persona'}), 404
```

Es lo mismo que:

```python
@personas_bp.route('/personas', methods=['GET'])
def obtener_personas():
    lista_diccionarios = []
    for p in repo_Personas.obtenerTodas():
        lista_diccionarios.append(p.toDiccionario())
    return jsonify(lista_diccionarios), 200
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

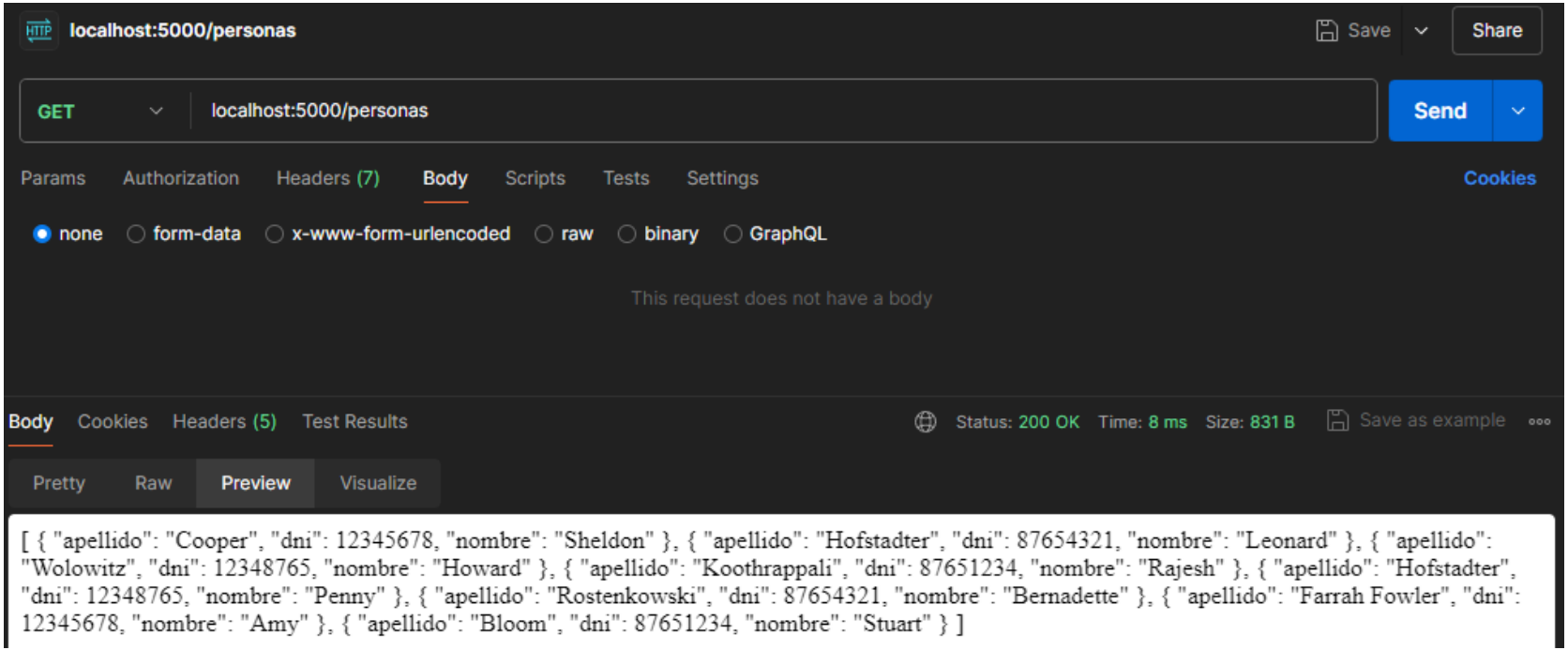
```python
@personas_bp.route('/personas', methods=['POST'])
def agregar_persona():
    if request.is_json:
        datos = request.get_json()
        try:
            persona = Persona.fromDiccionario(datos)
            repo_Personas.agregar(persona)
            return jsonify(persona.toDiccionario()), 201
        except ValueError as e:
            return jsonify({'error': str(e)}), 400
    else:
        return jsonify({'error': 'El contenido debe ser JSON'}), 400

@personas_bp.route('/personas/<int:dni>', methods=['PUT'])
def modificar_persona(dni):
    if request.is_json:
        datos = request.get_json()
        if "nombre" in datos and "apellido" in datos:
            nombre = datos['nombre']
            apellido = datos['apellido']
            if repo_Personas.modificarPorDni(dni, nombre, apellido):
                return jsonify({'mensaje': 'Persona modificada'}), 200
            else:
                return jsonify({'error': 'No se encontró la persona'}), 404
        else:
            return jsonify({'error': 'Faltan datos'}), 400
    else:
        return jsonify({'error': 'El contenido debe ser JSON'}), 400
```

```
mi_proyecto/
├── app.py
├── modelos/
│   ├── entidades/
│   │   └── persona.py
│   └── repos/
│       ├── repositorio_persona.py
│       └── repositorios.py
├── rutas/
│   └── rutas_persona.py
└── datos/
    └── persona.json
```

```
[
    {"dni": 12345678, "nombre": "Sheldon", "apellido": "Cooper"},
    {"dni": 87654321, "nombre": "Leonard", "apellido": "Hofstadter"},
    {"dni": 21348765, "nombre": "Howard", "apellido": "Wolowitz"},
    {"dni": 87652134, "nombre": "Rajesh", "apellido": "Koothrappali"},
    {"dni": 12438765, "nombre": "Penny", "apellido": "Hofstadter"},
    {"dni": 87654312, "nombre": "Bernadette", "apellido": "Rostenkowski"},
    {"dni": 21354678, "nombre": "Amy", "apellido": "Farrah Fowler"},
    {"dni": 86751234, "nombre": "Stuart", "apellido": "Bloom"},
    {"dni": 35795165, "nombre": "Wil", "apellido": "Wheaton"}
]
```

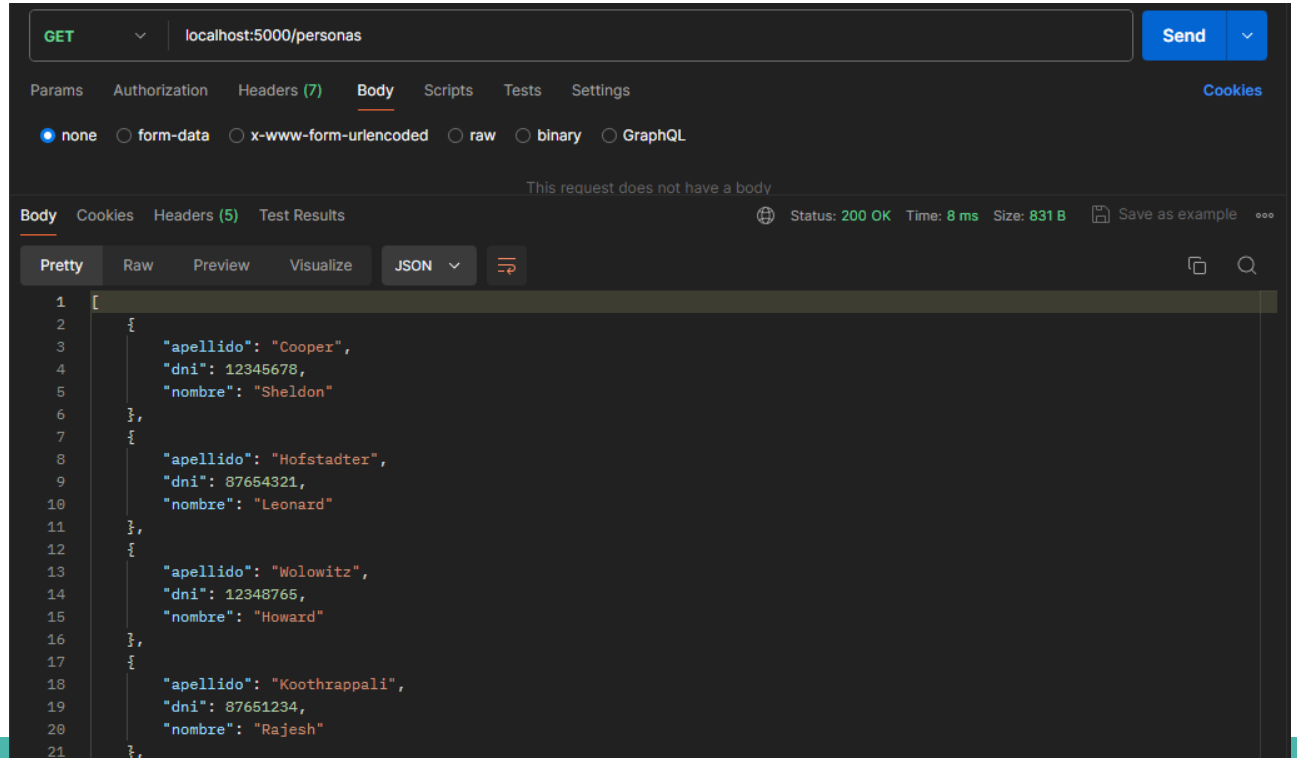
Probar las rutas con postman - GET

```python
@personas_bp.route('/personas', methods=['GET'])
def obtener_personas():
    return jsonify([p.toDiccionario() for p in repo_Personas.obtenerTodas()])
```



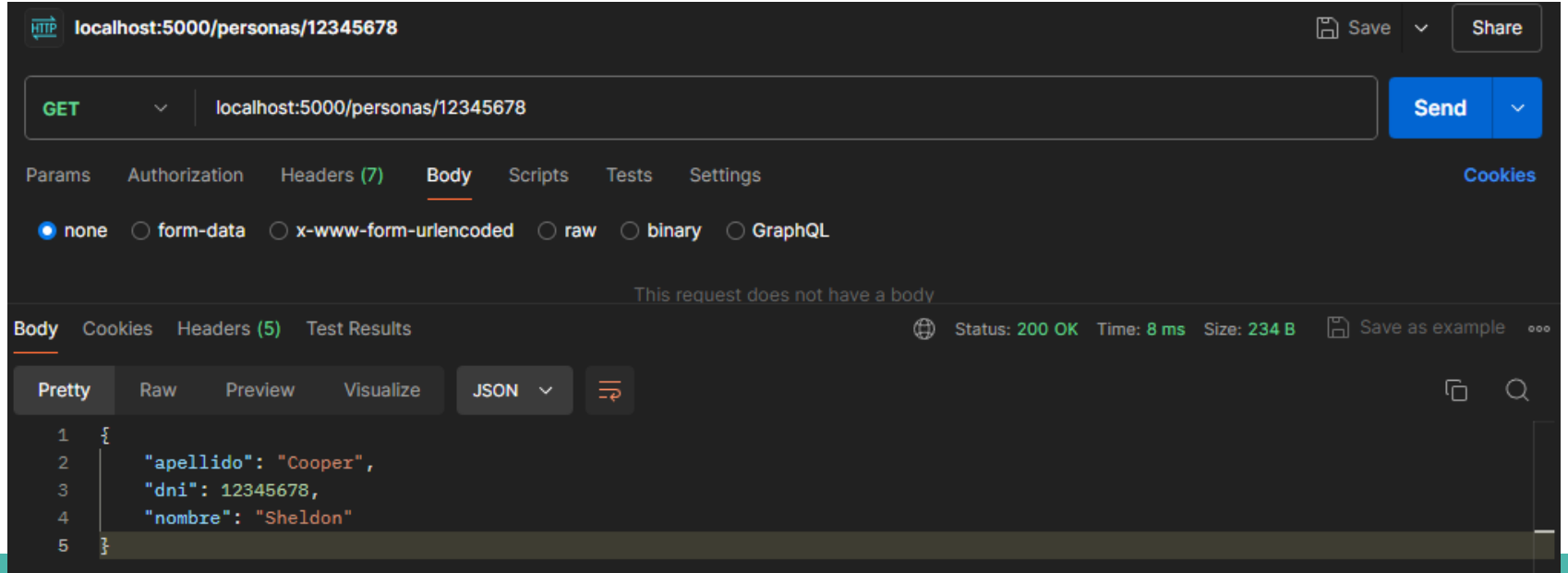
Probar las rutas con postman - GET

```python
@personas_bp.route('/personas', methods=['GET'])
def obtener_personas():
    return jsonify([p.toDiccionario() for p in repo_Personas.obtenerTodas()])
```



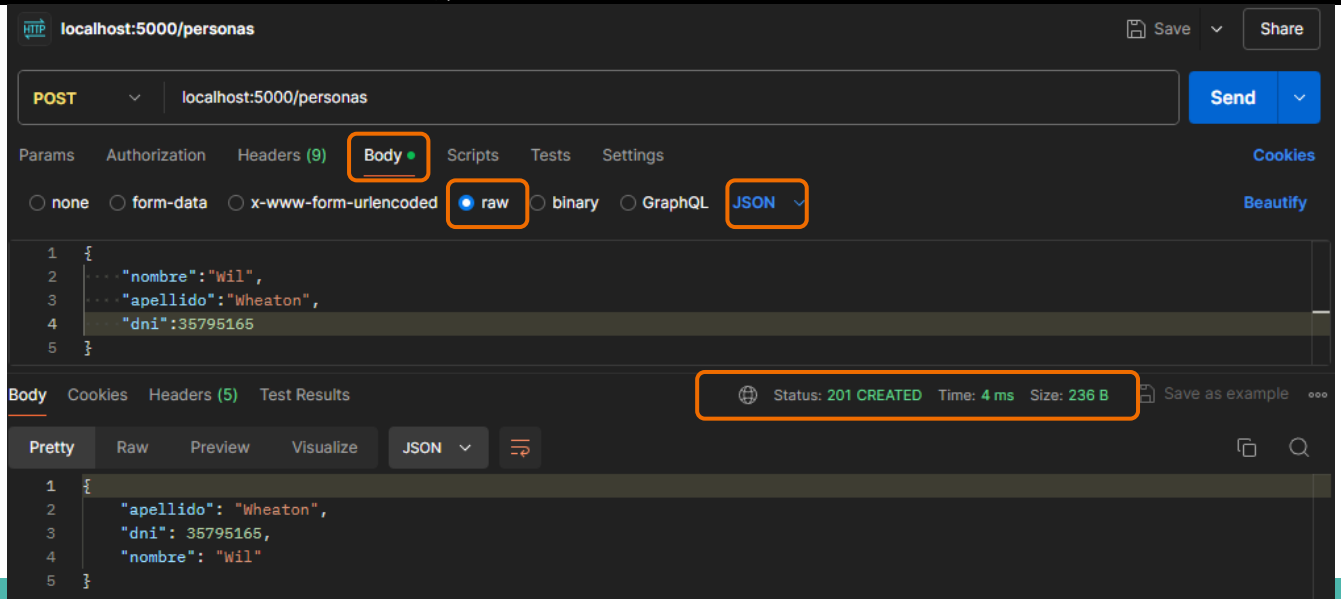
Probar las rutas con postman - GET por DNI

```python
@personas_bp.route('/personas/<int:dni>', methods=['GET'])
def obtener_persona(dni):
    persona_encontrada = repo_Personas.obtenerPorDni(dni)
    if isinstance(persona_encontrada, Persona):
        return jsonify(persona_encontrada.toDiccionario()), 200
    else:
        return jsonify({'error': 'No se encontró la persona'}), 404
```



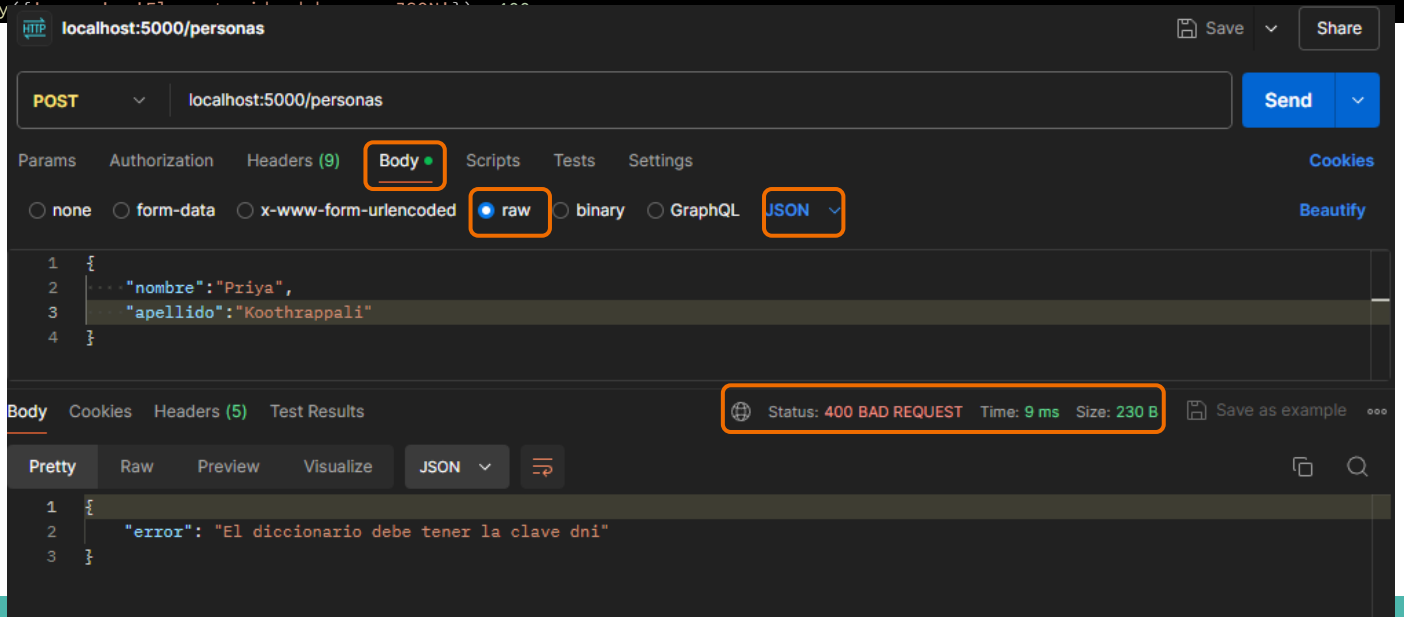
Probar las rutas con postman - POST

```python
@personas_bp.route('/personas', methods=['POST'])
def agregar_persona():
    if request.is_json:
        datos = request.get_json()
        try:
            persona = Persona.fromDiccionario(datos)
            repo_Personas.agregar(persona)
            return jsonify(persona.toDiccionario()), 201
        except ValueError as e:
            return jsonify({'error': str(e)}), 400
    else:
        return jsonify({'error': 'El contenido debe ser JSON'}), 400
```



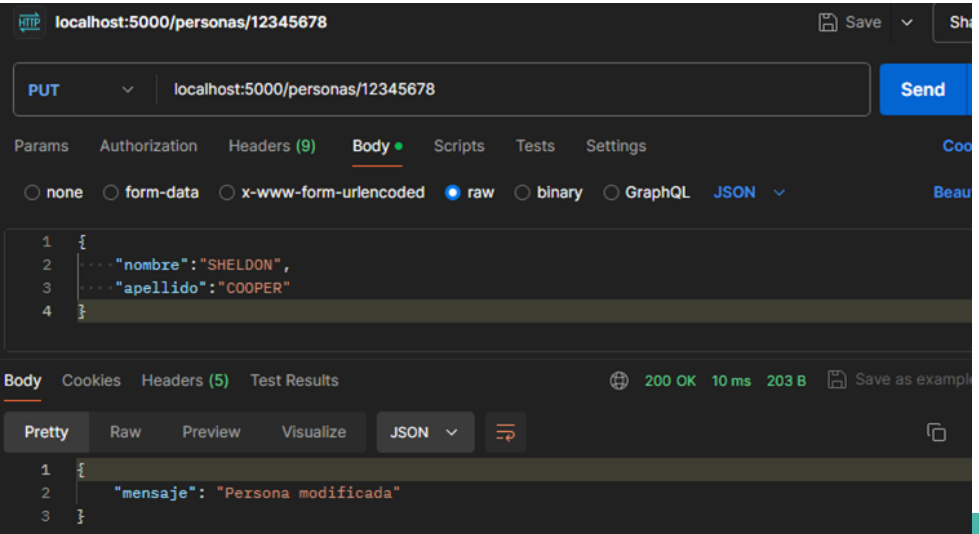
Probar las rutas con postman - POST

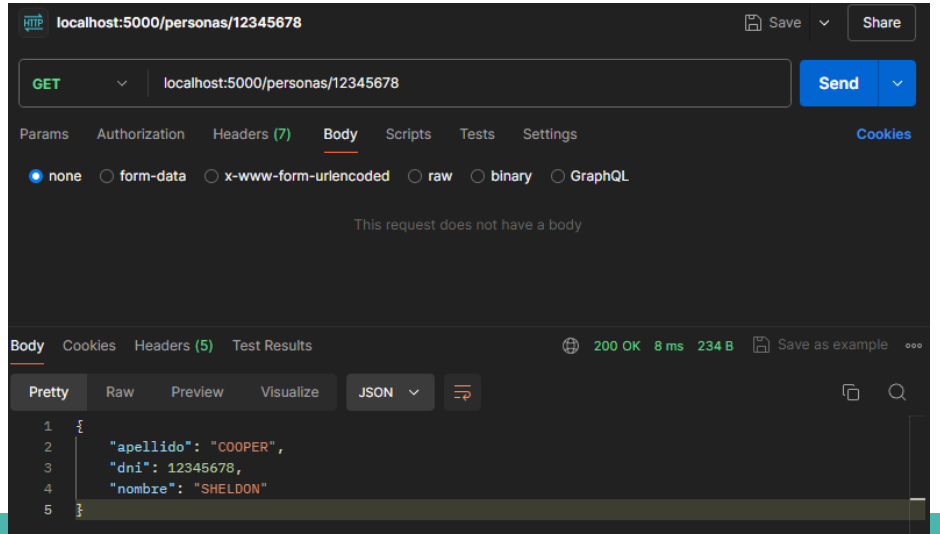
```python
@personas_bp.route('/personas', methods=['POST'])
def agregar_persona():
    if request.is_json:
        datos = request.get_json()
        try:
            persona = Persona.fromDiccionario(datos)
            repo_Personas.agregar(persona)
            return jsonify(persona.toDiccionario()), 201
        except ValueError as e:
            return jsonify({'error': str(e)}), 400
    else:
        return jsonify(
```



Probar las rutas con postman - PUT

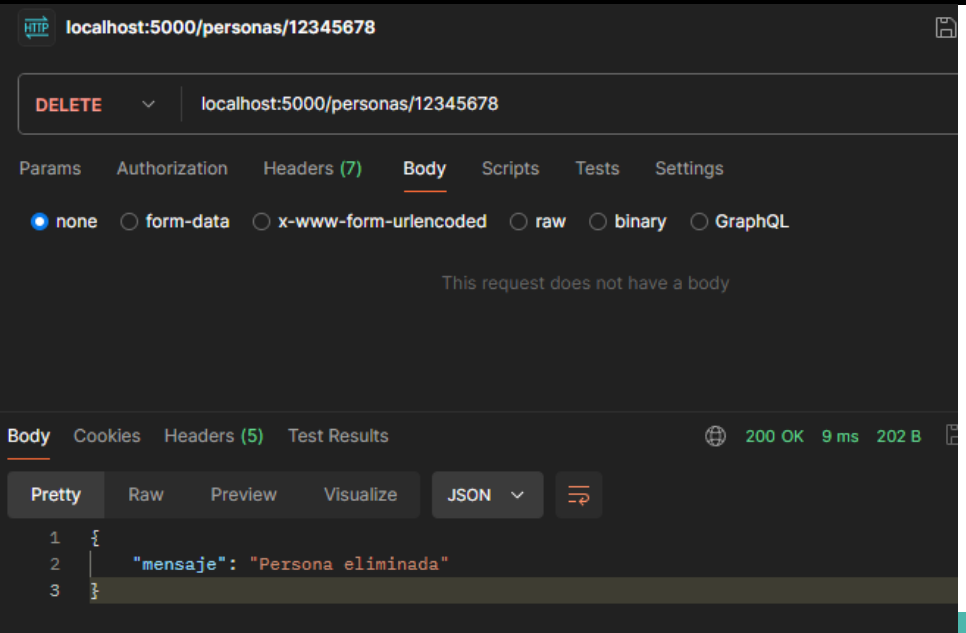
```python
@personas_bp.route('/personas/<int:dni>', methods=['PUT'])
def modificar_persona(dni):
    if request.is_json:
        datos = request.get_json()
        if "nombre" in datos and "apellido" in datos:
            nombre = datos['nombre']
            apellido = datos['apellido']
            if repo_Personas.modificarPorDni(dni, nombre, apellido):
                return jsonify({'mensaje': 'Persona modificada'}), 200
            else:
                return jsonify({'error': 'No se encontró la persona'}), 404
        else:
            return jsonify({'error': 'Faltan datos'}), 400
    else:
        return jsonify({'error': 'El contenido debe ser JSON'}), 400
```

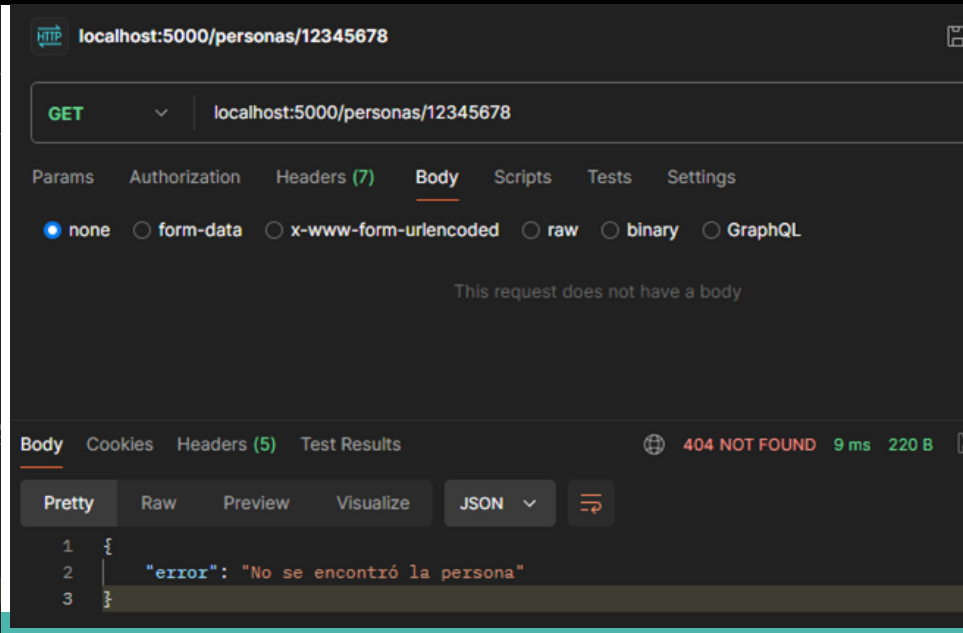




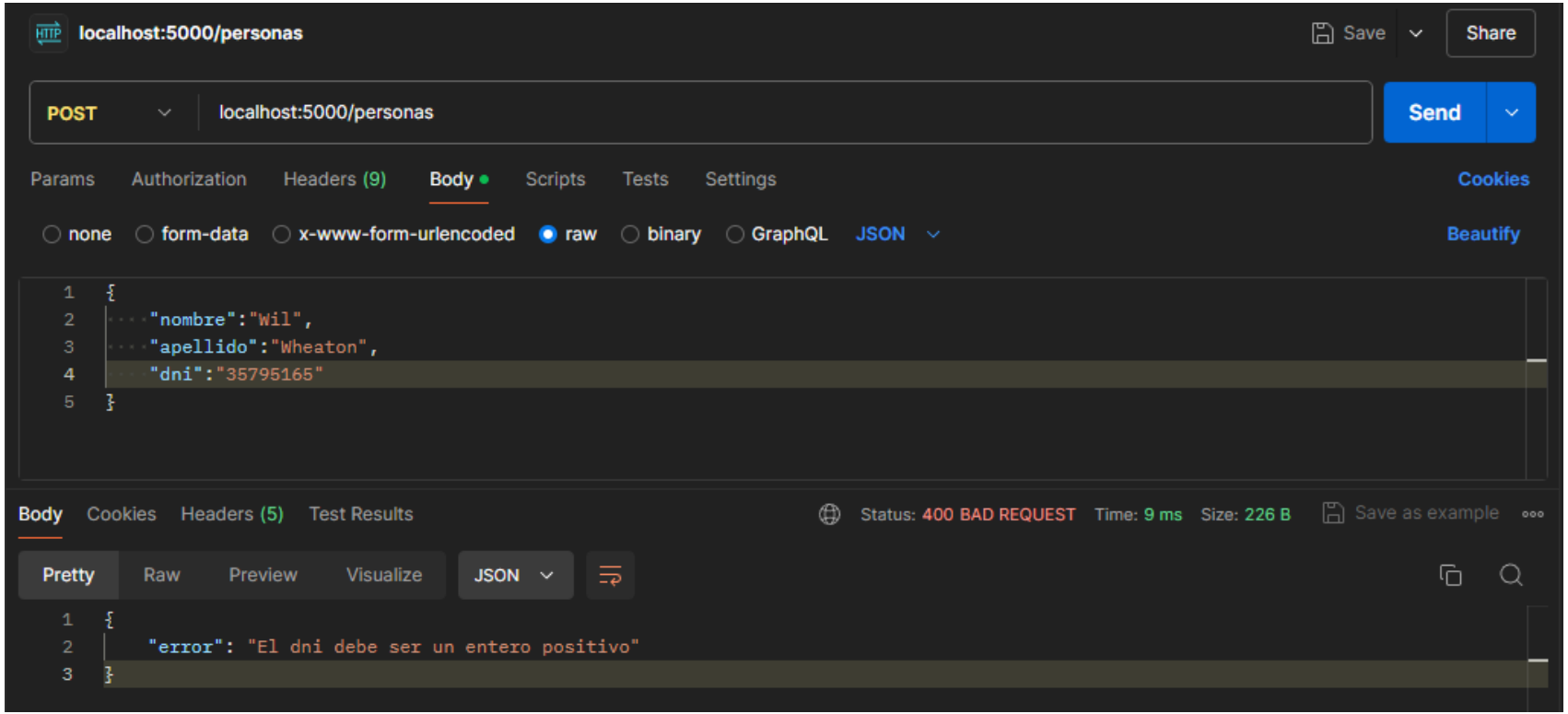
Probar las rutas con postman - DELETE

```python
@personas_bp.route('/personas/<int:dni>', methods=['DELETE'])
def eliminar_persona(dni):
    if repo_Personas.eliminarPorDni(dni):
        return jsonify({'mensaje': 'Persona eliminada'}), 200
    else:
        return jsonify({'error': 'No se encontró la persona'}), 404
```





Probar las rutas con postman - POST



Probar las rutas con postman - POST