

## Apuntes API-REST

### ¿Qué es una API REST?

Una API REST es una interfaz que expone *recursos* (libros, usuarios, productos, etc.) mediante URLs y utiliza el protocolo HTTP siguiendo estándares:

- **GET → obtener**
- **POST → crear**
- **PUT → actualizar**
- **DELETE → borrar**

Las respuestas se devuelven normalmente en JSON.

El cliente (Postman, frontend, app móvil) hace peticiones, y la API responde gestionando esos datos.

```
mi_api/
    └── app.py                                # Punto de entrada
    └── rutas/
        ├── ruta_usuarios.py                 # Controladores/Endpoints
        ├── ruta_productos.py
        └── ...
    └── modelos/
        ├── entidades/
            ├── usuario.py                  # Lógica de negocio
            ├── producto.py
            └── ...
        └── repos/
            ├── repositorio_usuarios.py
            ├── repositorio_productos.py
            └── repositorios.py           # Acceso a datos
                # Inicialización centralizada
    └── datos/
        ├── usuarios.json                # Almacenamiento JSON
        ├── productos.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)

Resumen:

- instanciar Flask
- registrar blueprints
- inicializar la app en modo debug=True (si cambian el código se refresca y da mensajes detallados de errores)

```
from flask import Flask
from rutas.rutasLibro import libros_bp

# Crear la instancia de la aplicación Flask
app = Flask(__name__)

# registrar el blueprint aqui
# Un blueprint es un conjunto de rutas (endpoints) agrupadas
# Esto conecta todas las rutas definidas en libros_bp a la app principal
app.register_blueprint(libros_bp)

# Este bloque solo se ejecuta si corremos directamente este archivo
if __name__ == "__main__":
    # Iniciar el servidor de desarrollo de Flask
    # debug=True activa:
    #     - Recarga automática cuando modificas el código
    #     - Mensajes de error detallados en el navegador
    #     - Modo desarrollo (NO usar en producción)
    app.run(debug=True)
```

## rutasLibro.py

(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.

Resumen:

- obtener la instancia del repositorioLibro para manejar el conjunto de datos
- crear un blueprint(agrupa todas las rutas)
- creamos las rutas (URL)
- jsonify nos permite retornar respuestas de la api en formato JSON, debemos agregar el código de respuesta, los que más usaremos serán 400 y 200

```
# Importar Blueprint para crear un conjunto de rutas agrupadas
# request para acceder a los datos de las peticiones HTTP
# jsonify para convertir datos de Python a formato JSON (respuesta de la API)
from flask import Blueprint, request, jsonify
from modelos.entidades.Libro import Libro

# Importar la función que devuelve la instancia del repositorio de libros
# El repositorio maneja el acceso a los datos (leer/escribir en JSON)
from modelos.repositorios.Repositorios import obtenerRepositorioLibro

# Obtener la instancia compartida del repositorio de libros
# Esta instancia se usará en todos los endpoints para acceder a los datos
repositorio_Libro = obtenerRepositorioLibro()

# Crear un Blueprint llamado 'libros_bp'
# Agrupa todas las rutas relacionadas con libros
# __name__ indica el módulo actual para que Flask lo identifique
libros_bp = Blueprint('libros_bp', __name__)

# Definir una ruta (endpoint) para obtener todos los libros
# @libros_bp.route: decorador que registra la función como endpoint
# '/libros': la URL del endpoint
# methods=['GET']: solo acepta peticiones GET (lectura de datos)
@libros_bp.route('/libros', methods=['GET'])
def obtener_libros():
    return jsonify([l.toDiccionario() for l in repositorio_Libro.obtenerTodos()])
```

## Repositorios.py

es útil para inicializar y centralizar las instancias de los repositorios, permitiendo que la aplicación use los mismos repositorios compartidos

Resumen:

- se asegura que solo se instancie una vez el repositorioLibro en este caso (evita usar listas duplicados)

```
# Importar la clase RepositorioLibros que maneja el acceso a datos
# Esta clase se encarga de leer/escribir libros en el archivo JSON
from modelos.repositorios.RepositorioLibros import RepositorioLibros

# Variable global que almacenará la instancia única del repositorio
# Inicialmente es None (no existe ninguna instancia)
repositorio_Libro = None

# Función que implementa el patrón Singleton para el repositorio de libros
# Garantiza que solo existe UNA instancia del repositorio en toda la aplicación
def obtenerRepositorioLibro():
    # Declarar que vamos a usar la variable global repositorio_Libro
    # Sin esto, Python crearía una variable local nueva
    global repositorio_Libro

    # Verificar si el repositorio aún no ha sido creado
    if repositorio_Libro is None:
        # Primera vez: crear la instancia del repositorio
        # Esto carga los datos del archivo JSON en memoria
        repositorio_Libro = RepositorioLibros()

    # Devolver la instancia del repositorio (nueva o existente)
    # Todas las rutas compartirán esta misma instancia
    return repositorio_Libro
```

## repositorioLibro.py

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.

Resumen:

- es una lista que guarda las entidades
- el RepositorioLibros lee libros.json (json.load) y lo convierte a objetos (con fromDiccionario) y los agrega a self.\_\_libros
- también RepositorioLibros guarda objetos (con toDiccionario) en el json (con json.dump)
- va a tener métodos que luego llamaremos en el archivo rutas\_libro.py

```
# Importar la clase Libro que representa la entidad de negocio
from modelos.entidades.Libro import Libro

# Importar el módulo json para leer y escribir archivos JSON
import json

# Clase Repositorio que maneja la persistencia de libros en archivo JSON
# Encapsula todas las operaciones de lectura/escritura de datos
class RepositorioLibros:
    # Constante de clase: ruta absoluta del archivo JSON donde se almacenan los libros
    # Es una variable de clase (compartida por todas las instancias)
    FILE_PATH = r"C:\Users\Pc\Desktop\TP9 2025\libros\datos\libros.json"

    # Constructor: se ejecuta al crear una instancia del repositorio
    def __init__(self):
        # Cargar todos los libros del archivo JSON a memoria
        # __libros es una lista privada (por convención) que almacena objetos Libro
        self.__libros = self.__cargarTodos()

    # Método privado para cargar todos los libros desde el archivo JSON
    # El doble guión bajo (__) indica que es un método privado (solo uso interno)
```

## entidades/Libro.py

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.

## datos/libros.json

Almacena los archivos JSON con los datos de las entidades

```
[  
  {  
    "isbn": 9788408266456,  
    "titulo": "El nombre del viento",  
    "autor": "Patrick Rothfuss",  
    "genero": "Fantasía",  
    "anio_publicacion": 2007,  
    "cantidad_ejemplares": 12  
  },  
  {  
    "isbn": 9789507317404,  
    "titulo": "El psicoanalista",  
    "autor": "John Katzenbach",  
    "genero": "Thriller psicológico",  
    "anio_publicacion": 2002,  
    "cantidad_ejemplares": 8  
  }  
]
```