

Universidad de Las Américas
Facultad de Ingenierías y Ciencias Agropecuarias
Ingeniería De Software
Progreso 1

Nombres: Sammy Porras, Daniel Vizcarra

Fecha: 13/11/2025

Diseño e Implementación de una API REST

Objetivo de la actividad:

El objetivo de este taller es diseñar, implementar y documentar una API REST utilizando un stack tecnológico flexible, cumpliendo los principios RESTful, la definición formal mediante OpenAPI 3.0, y las prácticas de integración descritas en el escenario del taller.}

1. Requisitos técnicos

Para el desarrollo del taller se seleccionó el stack **Python + FastAPI**, cumpliendo con la flexibilidad tecnológica permitida por el enunciado. Este stack cumple todas las condiciones solicitadas:

- Implementación de los endpoints requeridos (GET, POST y GET por ID).
- Generación automática de contrato OpenAPI 3.0.
- Ejecución reproducible mediante comandos simples (uvicorn main:app --reload).
- Documentación Swagger incorporada por defecto.
- Endpoint de salud /health.
- Colección de pruebas Postman.
- README técnico con instrucciones.

2. Paso a Paso para Desplegar la Solución

En esta sección se detalla el procedimiento seguido para diseñar, implementar y poner en funcionamiento la API REST utilizando **Python + FastAPI**, cumpliendo con las condiciones mínimas establecidas para el uso de un stack tecnológico alternativo, tales como la implementación de los endpoints requeridos, la provisión de un contrato OpenAPI 3.0 válido, la ejecución reproducible del servicio, el uso de una colección de pruebas en Postman y la exposición de un endpoint de salud /health.

2.1. Selección del stack tecnológico y preparación del entorno

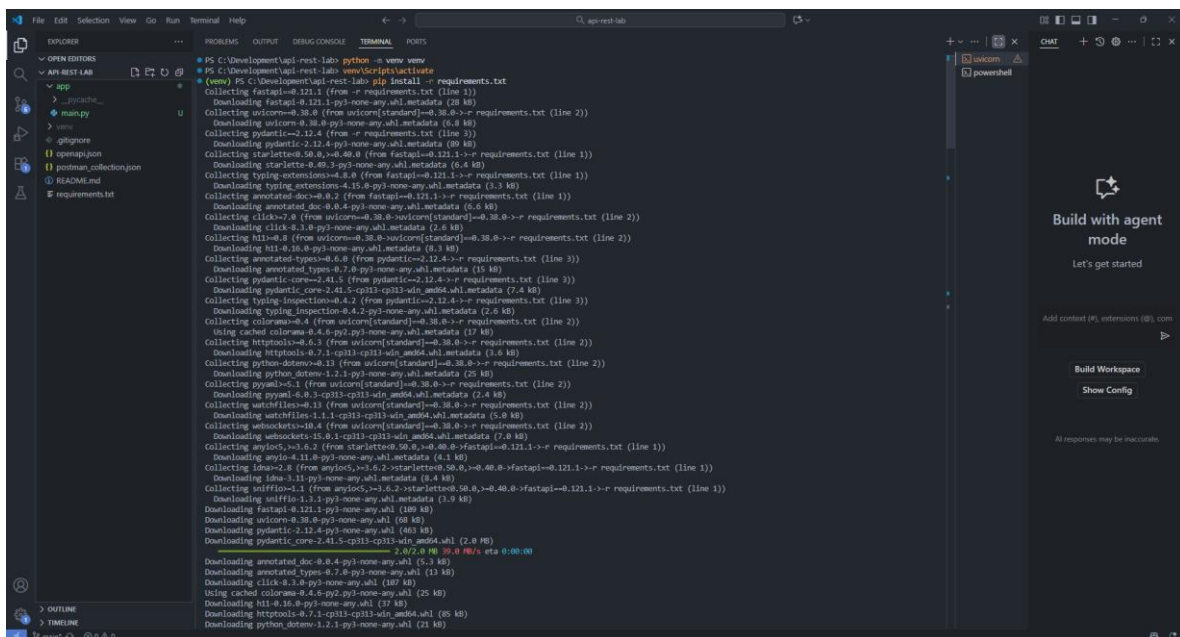
Como alternativa a la línea base propuesta en Java + Apache Camel, se eligió el stack **Python + FastAPI**, debido a las siguientes razones:

- FastAPI ofrece soporte nativo para **OpenAPI 3.0** y genera automáticamente la documentación Swagger.
- Permite implementar APIs REST de forma rápida, manteniendo buenas prácticas de diseño.
- La ejecución del servicio es reproducible mediante un solo comando (uvicorn), cumpliendo con las condiciones del taller.

Para preparar el entorno de trabajo se realizaron los siguientes pasos:

1. Creación de una carpeta de proyecto, por ejemplo: api-envios-fastapi/.
2. Creación y activación de un entorno virtual de Python para aislar las dependencias del proyecto:
3. python -m venv venv
4. # Windows
5. venv\Scripts\activate
6. # Linux/MacOS
7. source venv/bin/activate
8. Instalación de las dependencias necesarias:
9. pip install fastapi uvicorn

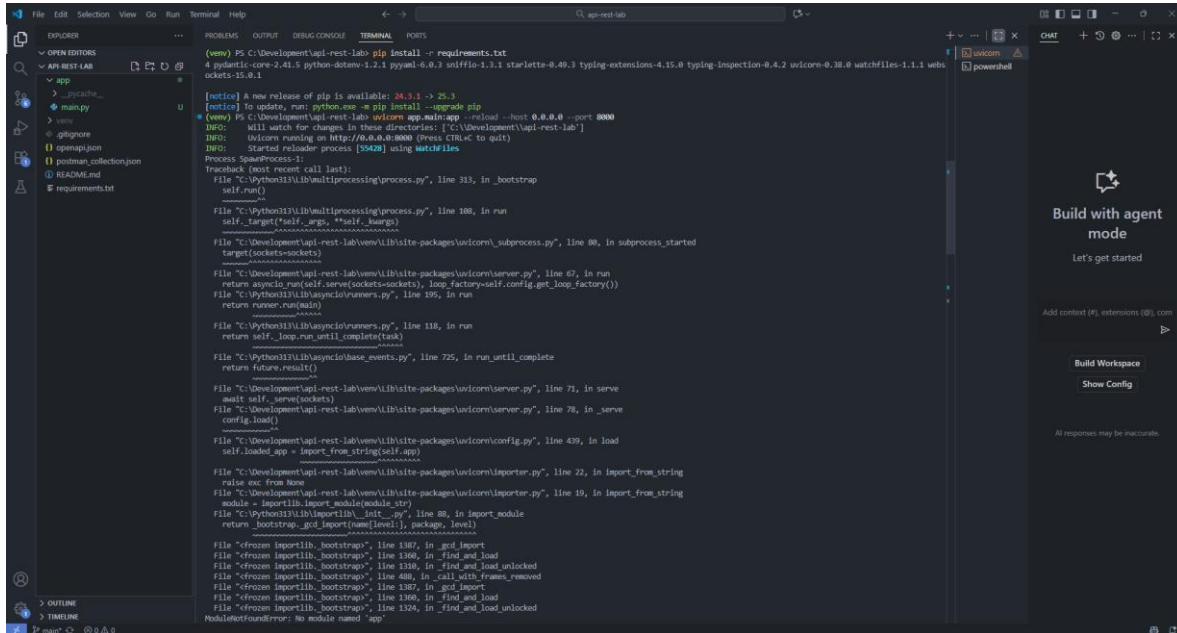
Con esto, el entorno quedó listo para iniciar la implementación de la API.



```

PS C:\Development\api-rest-lab> python -m venv venv
PS C:\Development\api-rest-lab> venv\Scripts\activate
Collecting fastapi==0.121.1 (from -r requirements.txt (line 1))
  Downloading fastapi-0.121.1-py3-none-any.whl.metadata (28 kB)
Collecting uvicorn==0.38.0 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading uvicorn-0.38.0-py3-none-any.whl.metadata (6.8 kB)
Collecting pydantic==2.12.4 (from -r requirements.txt (line 3))
  Downloading pydantic-2.12.4-py3-none-any.whl.metadata (80 kB)
Collecting starlette==0.49.0 (from fastapi==0.121.1->-r requirements.txt (line 1))
  Downloading starlette-0.49.3-py3-none-any.whl.metadata (6.4 kB)
Collecting typing_extensions==4.8 (from fastapi==0.121.1->-r requirements.txt (line 1))
  Downloading typing_extensions-4.8.0-py3-none-any.whl.metadata (3.3 kB)
Collecting annotated-doc==0.8.2 (from fastapi==0.121.1->-r requirements.txt (line 1))
  Downloading annotated-doc-0.8.4-py3-none-any.whl.metadata (5.6 kB)
Collecting click==7.0 (from uvicorn==0.38.0->uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading click-8.3.0-py3-none-any.whl.metadata (2.6 kB)
Collecting h11==0.8 (from uvicorn==0.38.0->uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading h11-0.16.0-py3-none-any.whl.metadata (8.3 kB)
Collecting annotated-types==0.6.0 (from pydantic==2.12.4->-r requirements.txt (line 3))
  Downloading annotated-types-0.7.0-py3-none-any.whl.metadata (15 kB)
Collecting pydantic_core==2.41.5 (from pydantic==2.12.4->-r requirements.txt (line 3))
  Downloading pydantic_core-2.41.5-cp313-cp313-win_amd64.whl.metadata (7.4 kB)
Collecting typing_inspection==0.4.2 (from pydantic==2.12.4->-r requirements.txt (line 3))
  Downloading typing_inspection-0.4.2-py3-none-any.whl.metadata (2.6 kB)
Collecting colorama==0.4 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Using cached colorama-0.4.4-py2.py3-none-any.whl.metadata (17 kB)
Collecting httpx==0.6.3 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading httpx-0.7.2-cp313-cp313-win_amd64.whl.metadata (3.8 kB)
Collecting python-dotenv==0.13 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading python_dotenv-1.2.1-py3-none-any.whl.metadata (2.4 kB)
Collecting pyyaml==6.0 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading pyyaml-6.0.3-cp313-cp313-win_amd64.whl.metadata (2.4 kB)
Collecting watchfiles==0.43 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading watchfiles-1.1.0-cp313-cp313-win_amd64.whl.metadata (5.8 kB)
Collecting websockets==15.0 (from uvicorn[standard]==0.38.0->-r requirements.txt (line 2))
  Downloading websockets-15.0.1-cp313-cp313-win_amd64.whl.metadata (7.8 kB)
Collecting anyio==5.0.0 (from starlette==0.49.0->uvicorn[standard]==0.38.0->-r requirements.txt (line 1))
  Downloading anyio-4.11.0-py3-none-any.whl.metadata (4.1 kB)
Collecting idna==2.8 (from anyio==5.0.0->starlette==0.49.0->uvicorn[standard]==0.38.0->-r requirements.txt (line 1))
  Downloading idna-3.11-py3-none-any.whl.metadata (8.4 kB)
Collecting sniffio==1.1 (from anyio==5.0.0->starlette==0.49.0->-r requirements.txt (line 1))
  Downloading sniffio-1.3.1-py3-none-any.whl.metadata (3.9 kB)
Collecting fastapi==0.121.1 (from -r requirements.txt (line 1))
  Downloading fastapi-0.121.1-py3-none-any.whl (109 kB)
Collecting uvicorn==0.38.0 (from -r requirements.txt (line 1))
  Downloading uvicorn-0.38.0-py3-none-any.whl (68 kB)
Collecting pydantic==2.12.4 (from -r requirements.txt (line 1))
  Downloading pydantic-2.12.4-py3-none-any.whl (463 kB)
Collecting pydantic_core==2.41.5 (from pydantic==2.12.4->-r requirements.txt (line 1))
  Downloading pydantic_core-2.41.5-cp313-cp313-win_amd64.whl (2.8 MB)
Collecting annotated-doc==0.8.4 (from fastapi==0.121.1->-r requirements.txt (line 1))
  Downloading annotated-doc-0.8.4-py3-none-any.whl (5.3 kB)
Collecting annotated-types==0.7.0 (from pydantic==2.12.4->-r requirements.txt (line 1))
  Downloading annotated-types-0.7.0-py3-none-any.whl (13 kB)
Collecting click==7.0 (from uvicorn==0.38.0->-r requirements.txt (line 1))
  Downloading click-8.3.0-py3-none-any.whl (107 kB)
Using cached colorama-0.4.4-py2.py3-none-any.whl (25 kB)
Collecting h11==0.16.0 (from uvicorn==0.38.0->-r requirements.txt (line 1))
  Downloading h11-0.16.0-py3-none-any.whl (37 kB)
Collecting httpx==0.7.2 (from uvicorn==0.38.0->-r requirements.txt (line 1))
  Downloading httpx-0.7.2-cp313-cp313-win_amd64.whl (85 kB)
Collecting python_dotenv==1.2.1 (from uvicorn==0.38.0->-r requirements.txt (line 1))
  Downloading python_dotenv-1.2.1-py3-none-any.whl (21 kB)

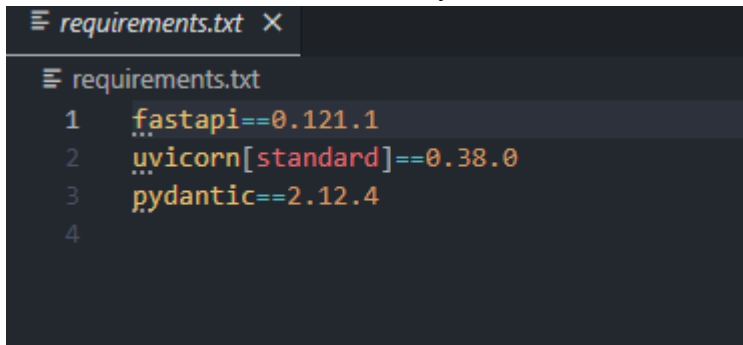
```



```
(venv) PS C:\Development\api-rest-lab pip install -r requirements.txt
4 pydantic-core-2.41.5 python-dotenv-1.2.1 pyyaml-6.0.3 sniffio-1.3.1 starlette-0.40.3 typing-extensions-4.15.0 typing-inspection-0.4.2 uvicorn-0.38.0 watchfiles-1.1.1 webs
ockets-15.0.1

[notice] A new release of pip is available: 24.3.1 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Development\api-rest-lab uvicorn app:main:app --reload --host 0.0.0.0 --port 8000
INFO: Will watch for changes in these directories: ['C:\Development\api-rest-lab']
INFO: Uvicorn running on http://0.0.0.0:8000 (Press Ctrl+C to quit)
INFO: Started reload process [50428] using watchfiles
Process SpawnProcess-1:
Traceback (most recent call last):
  File "C:\Python311\Lib\multiprocessing\process.py", line 313, in _bootstrap
    self.run()
  File "C:\Python311\Lib\multiprocessing\process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\subprocess.py", line 86, in subprocess_started
    target(sockets=sockets)
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\server.py", line 67, in run
    return asyncio.run(self.serve(sockets=sockets), loop_factory=self.config.get_loop_factory())
  File "C:\Python311\Lib\asyncio\runners.py", line 195, in run
    return runner.run(main)
  File "C:\Python311\Lib\asyncio\runners.py", line 118, in run
    return self._loop.run_until_complete(task)
  File "C:\Python311\Lib\asyncio\base_events.py", line 725, in run_until_complete
    return future.result()
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\server.py", line 71, in serve
    await self._serve(sockets)
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\server.py", line 78, in _serve
    config.load()
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\config.py", line 439, in load
    self.loaded_app = import_from_string(self.app)
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\importer.py", line 22, in import_from_string
    raise exc from None
  File "C:\Development\api-rest-lab\venv\Lib\site-packages\uvicorn\importer.py", line 19, in import_from_string
    module = importlib.import_module(module_str)
  File "C:\Python311\Lib\importlib\__init__.py", line 88, in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
  File "<frozen importlib._bootstrap>", line 1187, in _gcd_import
  File "<frozen importlib._bootstrap>", line 1166, in _find_and_load
  File "<frozen importlib._bootstrap>", line 1138, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 488, in _call_with_frames_removed
  File "<frozen importlib._bootstrap>", line 1187, in _gcd_import
  File "<frozen importlib._bootstrap>", line 1166, in _find_and_load
  File "<frozen importlib._bootstrap>", line 1134, in _find_and_load_unlocked
ModuleNotFoundError: No module named 'app'
```

Creamos un entorno virtual con Python e instalamos las dependencias de requirements.txt



```
requirements.txt
1 fastapi==0.121.1
2 uvicorn[standard]==0.38.0
3 pydantic==2.12.4
4
```

Con esto ya tenemos uvicorn para realizar las pruebas.

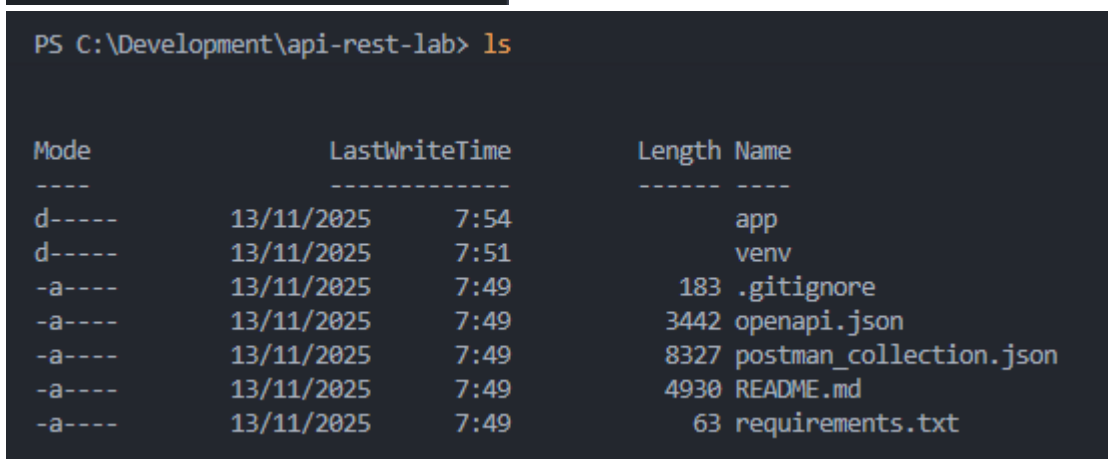
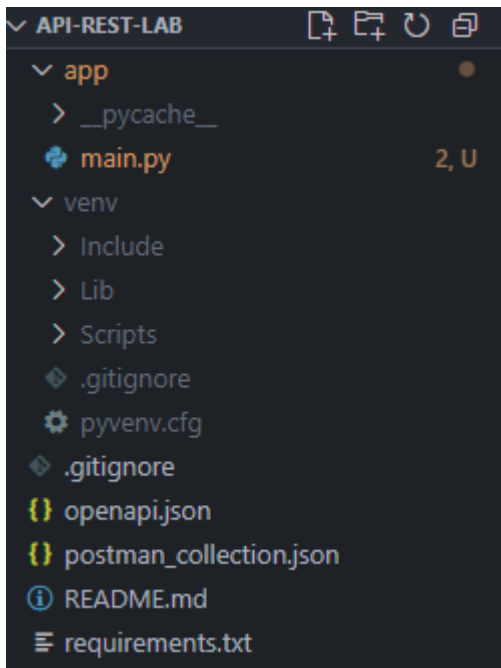
2.2. Estructura del proyecto

Dentro de la carpeta del proyecto se creó el archivo principal main.py, donde se concentra la lógica de la API. La estructura mínima inicial fue la siguiente:

api-envios-fastapi/

```
├── venv/
├── main.py
└── README.md
```

Esta organización permite mantener un proyecto sencillo pero ordenado, sobre el cual se pueden añadir más módulos o paquetes en caso de que la API crezca.



La estructura del proyecto tal como se planteó

2.3. Definición del modelo de dominio

Para representar los datos de un envío, se definió un modelo utilizando **Pydantic**, el cual permite validar de forma automática la estructura de los datos de entrada y salida:

- **Modelo Envío:** contiene atributos como id, destinatario, direccion y estado. Este modelo se utilizó tanto en las respuestas de los endpoints como en el cuerpo del POST /envios, garantizando consistencia en la representación de los datos. Adicionalmente, se implementó una estructura de datos en memoria (envios_db) que simula una base de datos, permitiendo realizar pruebas sin necesidad de un motor de base de datos real.

```
class Envio(BaseModel):
    id: int
    destinatario: str
    direccion: str
    estado: str

class Config:
    json_schema_extra = {
        "example": {
            "id": 1,
            "destinatario": "Juan Pérez",
            "direccion": "Av. República 123, Quito",
            "estado": "En tránsito"
        }
    }

envios_db: List[Envio] = [
    Envio(id=1, destinatario="María García", direccion="Calle 10 de Agosto 456, Quito", estado="Entregado"),
    Envio(id=2, destinatario="Carlos López", direccion="Av. Amazonas 789, Quito", estado="En tránsito"),
    Envio(id=3, destinatario="Ana Rodríguez", direccion="Calle Colón 321, Quito", estado="Pendiente")
]
```

Definición de la clase Envio con sus campos (id, destinatario, direccion, estado) y la lista envios_db con datos de ejemplo

2.4. Implementación de los endpoints REST

Se implementaron los tres endpoints solicitados en el enunciado del taller, cumpliendo con las operaciones básicas de consulta y creación de envíos:

1. GET /envios

- Retorna la lista completa de envíos almacenados en envios_db.
- Se declaró con tipo de respuesta List[Envio] para garantizar una salida tipada.

2. GET /envios/{id}

- Recibe un identificador de envío como parámetro de ruta.
- Recorre la colección envios_db para encontrar el envío correspondiente.
- Si lo encuentra, devuelve sus datos; en caso contrario, retorna un mensaje de error simple (en una mejora futura podría devolverse un código de estado 404).

3. POST /envios

- Recibe un objeto Envio en el cuerpo de la petición.
- Añade el envío a la colección envios_db.
- Retorna un mensaje de confirmación y un código de estado 201 Created.

Cada endpoint fue diseñado respetando los principios REST:

- Uso de métodos HTTP semánticos (GET, POST).
- Uso de rutas orientadas a recursos (/envios, /envios/{id}).
- Manejo de representaciones en formato JSON.

Declaración de cada endpoint en main.py.

```
# GET /envios - Obtener todos los envíos
@app.get("/envios", response_model=List[Envio], tags=["Envíos"])
def obtener_envios():
    """
    Retorna la lista completa de envíos almacenados en el sistema
    """
    return envios_db

# GET /envios/{id} - Obtener un envío por ID
@app.get("/envios/{id}", response_model=Envio, tags=["Envíos"])
def obtener_envio_por_id(id: int):
    """
    Retorna un envío específico basado en su ID

    - **id**: Identificador único del envío
    """
    # Buscar el envío en la base de datos
    for envio in envios_db:
        if envio.id == id:
            return envio

    # Si no se encuentra, retornar error 404
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail=f"Envío con ID {id} no encontrado"
    )

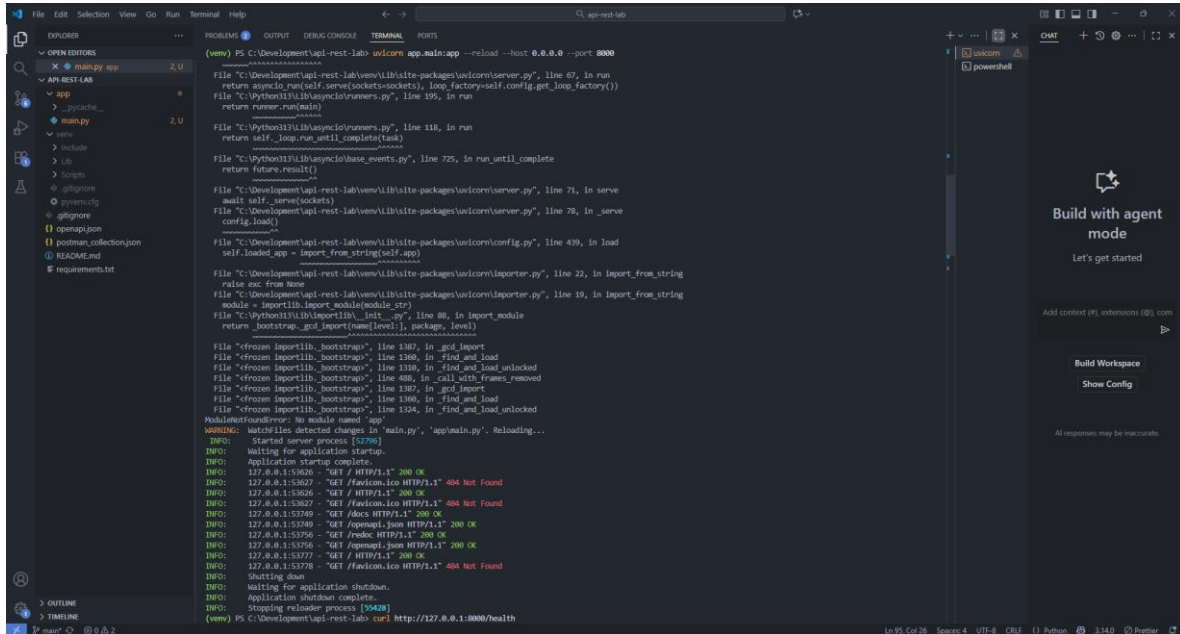
# POST /envios - Crear un nuevo envío
@app.post("/envios", response_model=Envio, status_code=status.HTTP_201_CREATED, tags=["Envíos"])
def crear_envio(envio: Envio):
    """
    Crea un nuevo envío en el sistema

    - **id**: Identificador único del envío
    - **destinatario**: Nombre del destinatario
    - **direccion**: Dirección de entrega
    - **estado**: Estado actual del envío (Pendiente, En tránsito, Entregado)
    """
    # Verificar si ya existe un envío con ese ID
    for env in envios_db:
        if env.id == envio.id:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"Ya existe un envío con ID {envio.id}"
            )

    # Agregar el envío a la base de datos
    envios_db.append(envio)

    return envio
```

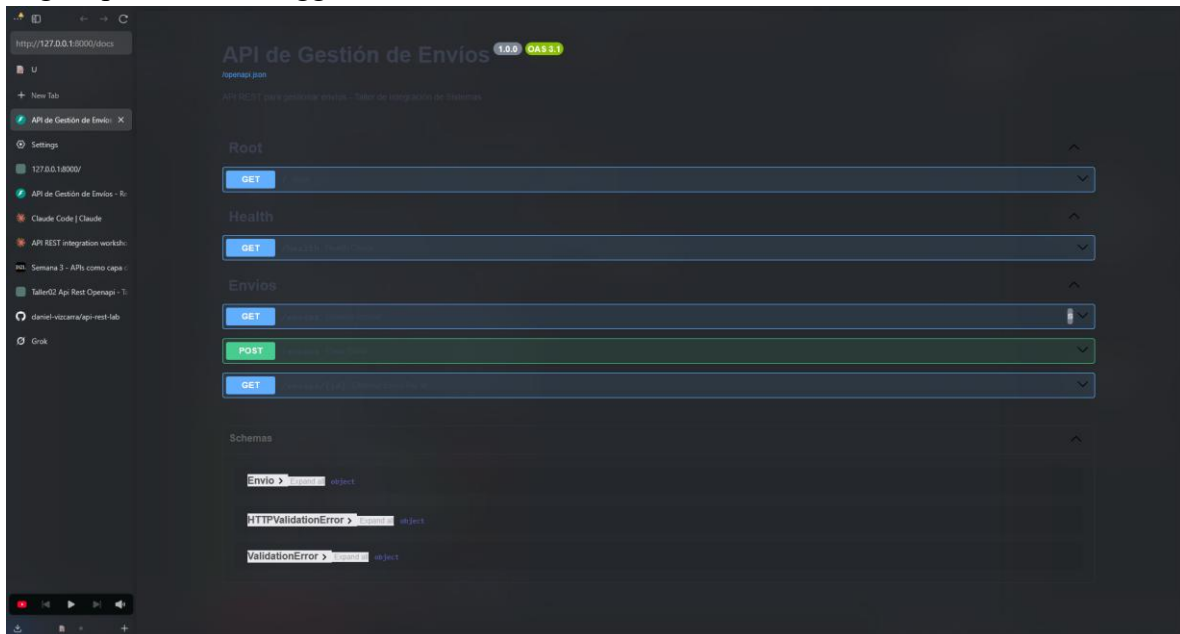
Levantamos el servidor y accedemos a la url <http://127.0.0.1:8000/docs> para acceder a swagger.



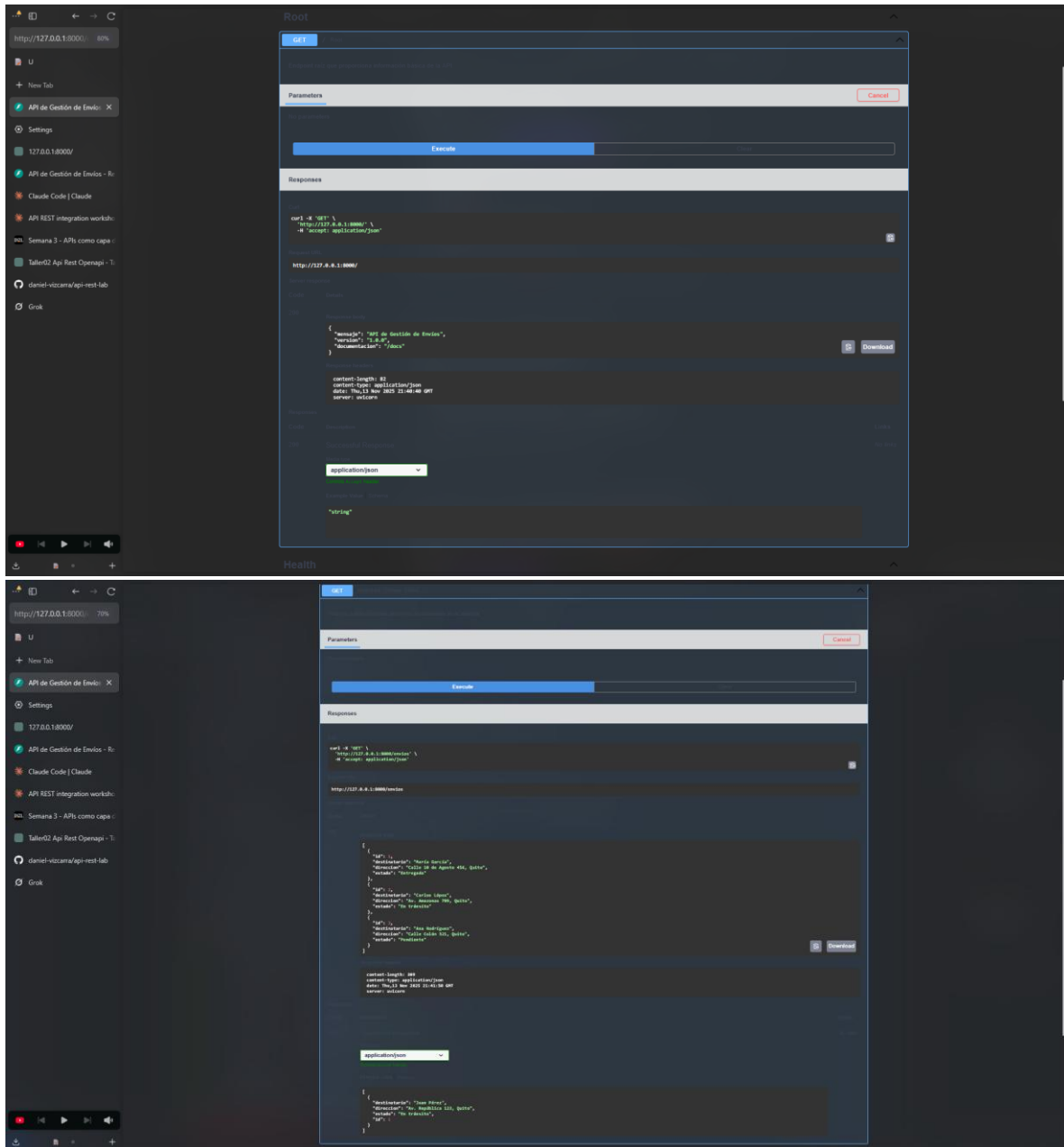
```

(venv) PS C:\Development\api-rest-lab\udicom app> main.py --reload --host 0.0.0.0 --port 8000
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\server.py", line 67, in run
    return asyncio_run(self.serve(sockets=sockets), loop_factory=self.config.get_loop_factory())
File "C:\Python11\lib\asyncio\runners.py", line 195, in run
    return runner.run(main)
File "C:\Python11\lib\asyncio\runners.py", line 118, in run
    return self._loop.run_until_complete(task)
File "C:\Python11\lib\asyncio\base_events.py", line 725, in run_until_complete
    return future.result()
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\server.py", line 71, in serve
    await self.serve(sockets)
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\server.py", line 78, in _serve
    config.load()
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\config.py", line 439, in load
    self.loaded_app = import_from_string(self.app)
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 22, in import_from_string
    raise exc from None
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 19, in import_from_string
    module = importlib.import_module(module_str)
File "C:\Python11\lib\importlib\_bootstrap.py", line 88, in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 1307, in _gcd_import
    File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 1306, in _find_and_load
    File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 1306, in _find_and_load_unlocked
    File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 448, in _call_with_frames_removed
    File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 1307, in _gcd_import
    File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 1306, in _find_and_load
    File "C:\Development\api-rest-lab\venv\lib\site-packages\urllib\importer.py", line 1306, in _find_and_load_unlocked
ModuleNotFoundError: No module named 'app'
WARNING: ModuleNotFoundError detected changes in 'main.py', 'app\main.py'. Reloading...
INFO: Started server process [5276]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:53626 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:53626 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:53627 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:53749 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:53749 - "GET /api/swagger.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:53756 - "GET /redoc HTTP/1.1" 200 OK
INFO: 127.0.0.1:53756 - "GET /api/swagger.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:53772 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:53778 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: Shutting down
INFO: Waiting for application shutdown.
INFO: Application shutdown complete.
INFO: Stopping reloader process [53428]
(venv) PS C:\Development\api-rest-lab> curl http://127.0.0.1:8000/health
  
```

Página principal de swagger



Realizamos pruebas de diferentes endpoints con Swagger:



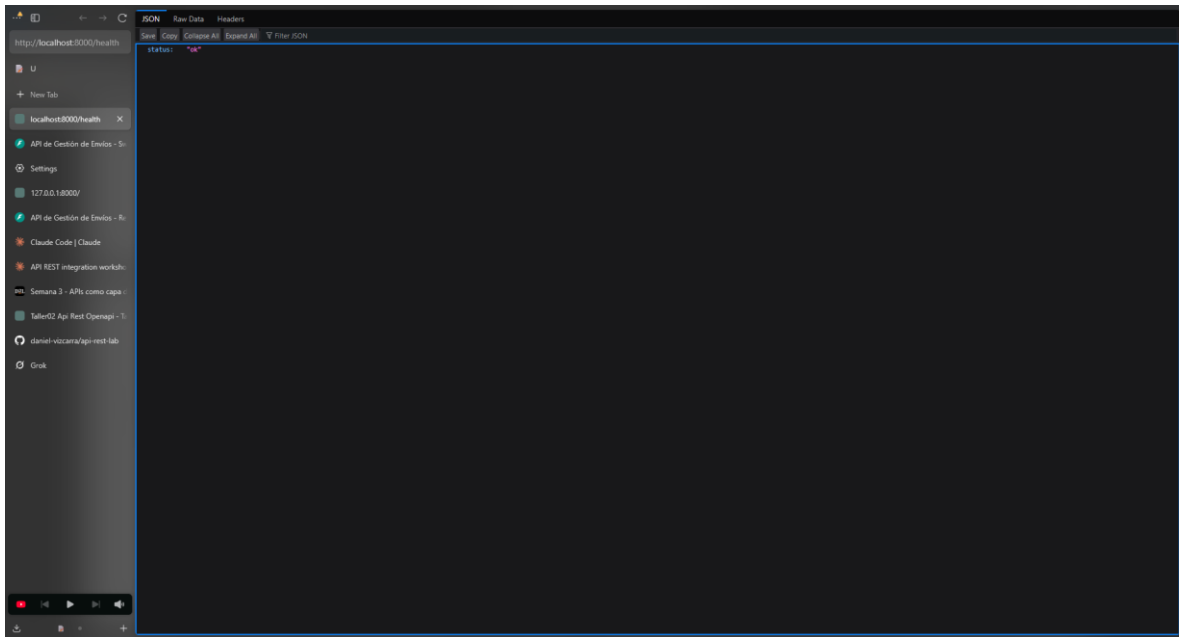
2.5. Endpoint de salud /health

En cumplimiento de los requisitos del stack libre, se añadió un endpoint de salud:

- **GET /health**
 - Devuelve un objeto JSON sencillo, por ejemplo: `{"status": "ok"}`.
 - Permite verificar rápidamente que la aplicación está desplegada y respondiendo, sin necesidad de invocar los endpoints funcionales.

Este endpoint es especialmente útil en escenarios de monitoreo o despliegue en contenedores, donde herramientas externas necesitan verificar el estado del servicio.

Accedemos a <http://localhost:8000/health> y verificamos que el endpoint Health esté correcto, debería mostrar OK.



2.6. Documentación automática con Swagger/OpenAPI

Una de las ventajas principales de FastAPI es que genera automáticamente la documentación OpenAPI 3.0, cumpliendo así el requisito de incluir un contrato válido (openapi.json).

Al ejecutar la aplicación, se exponen las siguientes URLs:

- **Documentación interactiva (Swagger UI):**
<http://127.0.0.1:8000/docs>
- **Contrato OpenAPI 3.0 en formato JSON:**
<http://127.0.0.1:8000/openapi.json>

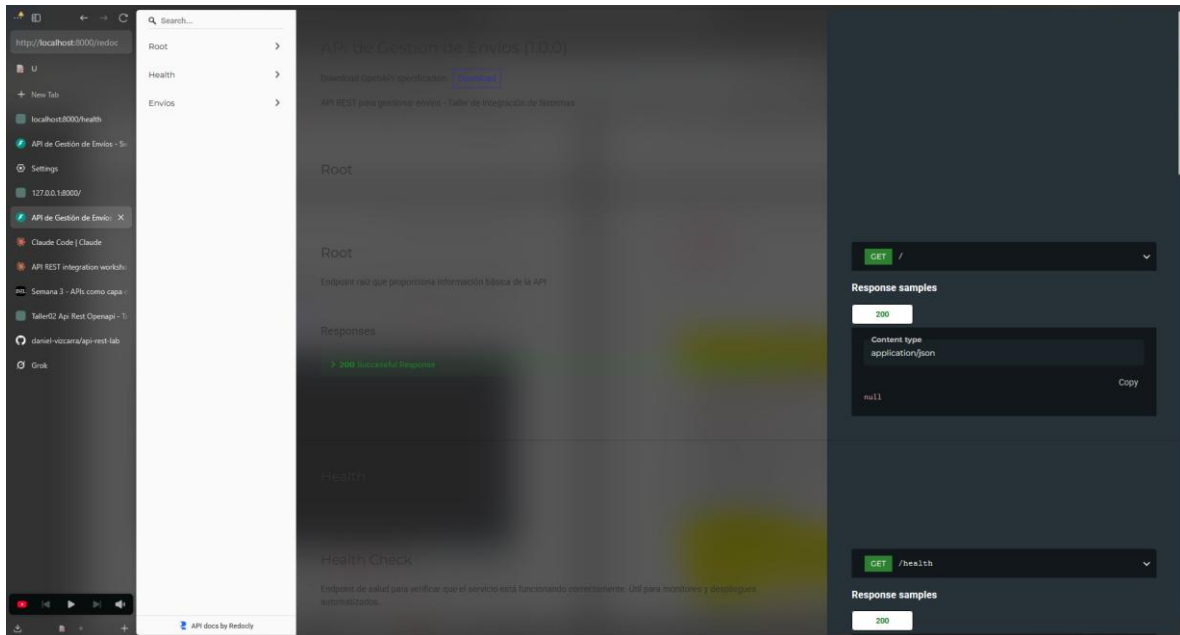
Desde Swagger UI fue posible:

- Visualizar todos los endpoints definidos.
- Consultar las estructuras de entrada y salida.
- Probar los métodos de forma interactiva (GET y POST) sin necesidad de herramientas externas.

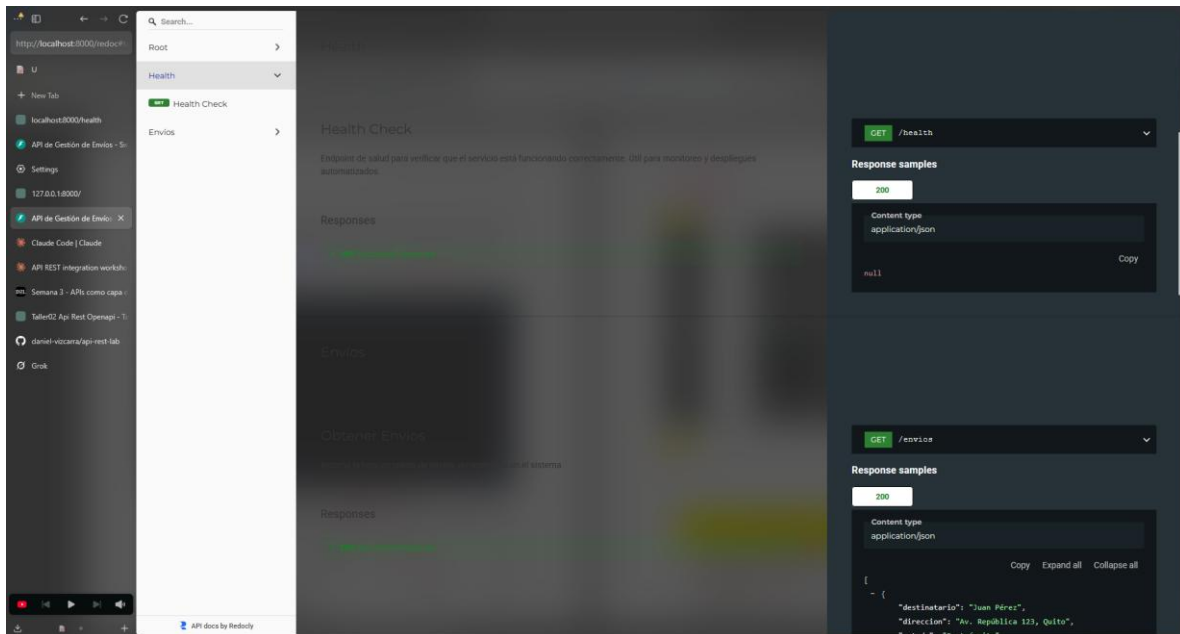
El archivo openapi.json se descargó y se incluyó dentro del repositorio como parte del contrato formal de la API.

Listamos los endpoints ahora en OpenApi:

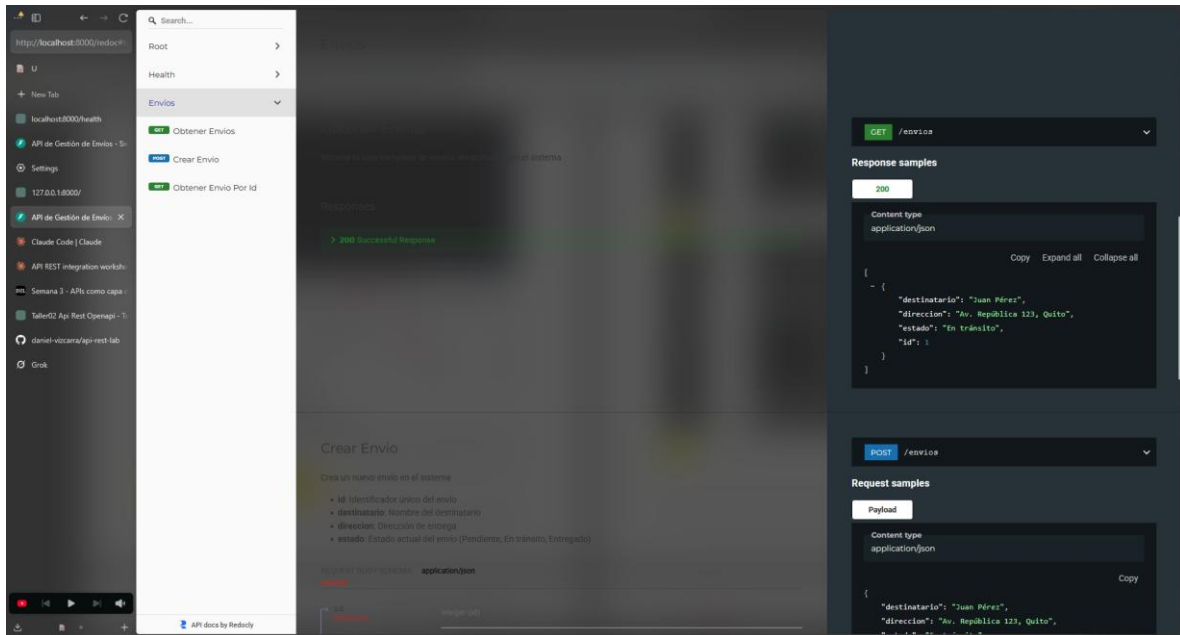
ROOT



Health



Envíos



2.7. Ejecución reproducible de la API

Para garantizar que cualquier integrante del equipo, docente o evaluador pueda ejecutar la API de forma reproducible, se definió el siguiente comando estándar:

uvicorn main:app --reload

Este comando:

- Levanta el servidor en `http://127.0.0.1:8000`.
- Permite recargar automáticamente la aplicación ante cambios en el código (`--reload`).

En el README técnico se documentaron los pasos necesarios:

1. Clonar el repositorio.
2. Crear y activar el entorno virtual.
3. Instalar las dependencias.
4. Ejecutar el comando de uvicorn.

Consola con el servidor corriendo y el log de las peticiones realizadas.

```
(venv) PS C:\Development\api-rest-lab> uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
INFO: Will watch for changes in these directories: ['C:\\Development\\api-rest-lab']
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO: Started reloader process [32732] using WatchFiles
INFO: Started server process [40180]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:62091 - "GET /redoc HTTP/1.1" 200 OK
INFO: 127.0.0.1:62091 - "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:62095 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:62095 - "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:62095 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:62097 - "GET /envios HTTP/1.1" 200 OK
```

2.8. Pruebas con Postman y generación de colección

Además de las pruebas realizadas desde Swagger, se construyó una colección de Postman con los siguientes endpoints:

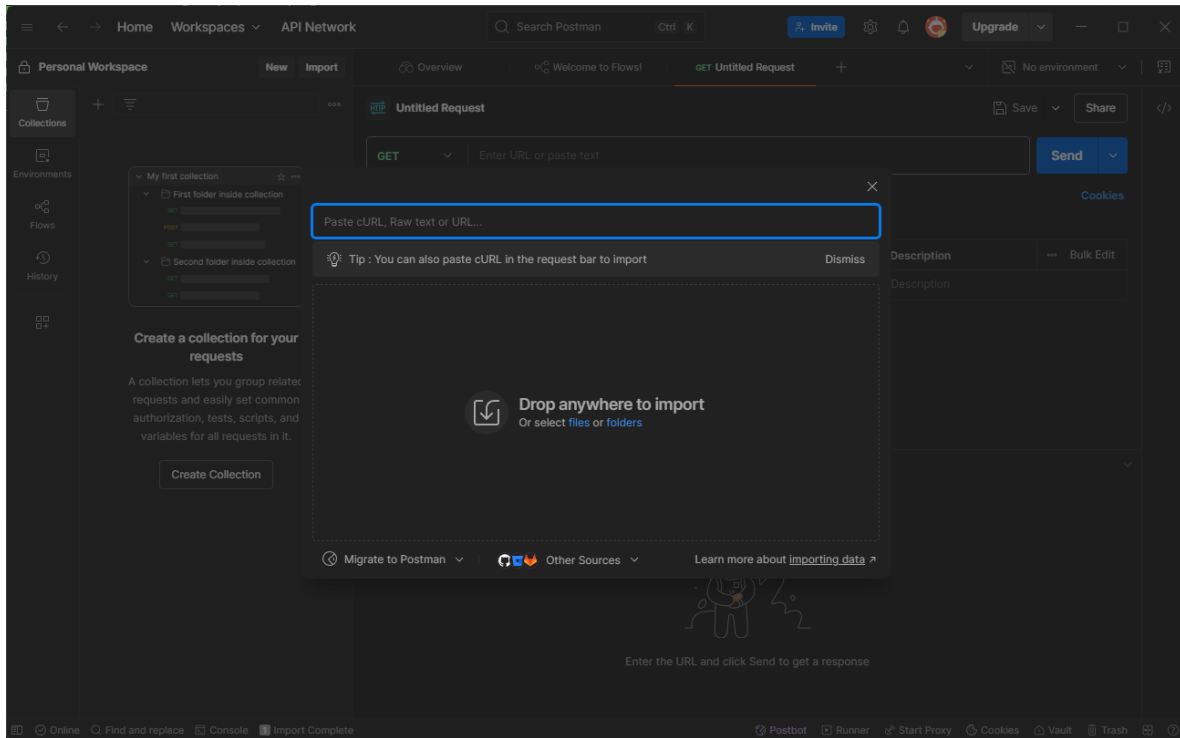
- GET /envios
- GET /envios/{id}
- POST /envios
- GET /health

Cada petición fue configurada con:

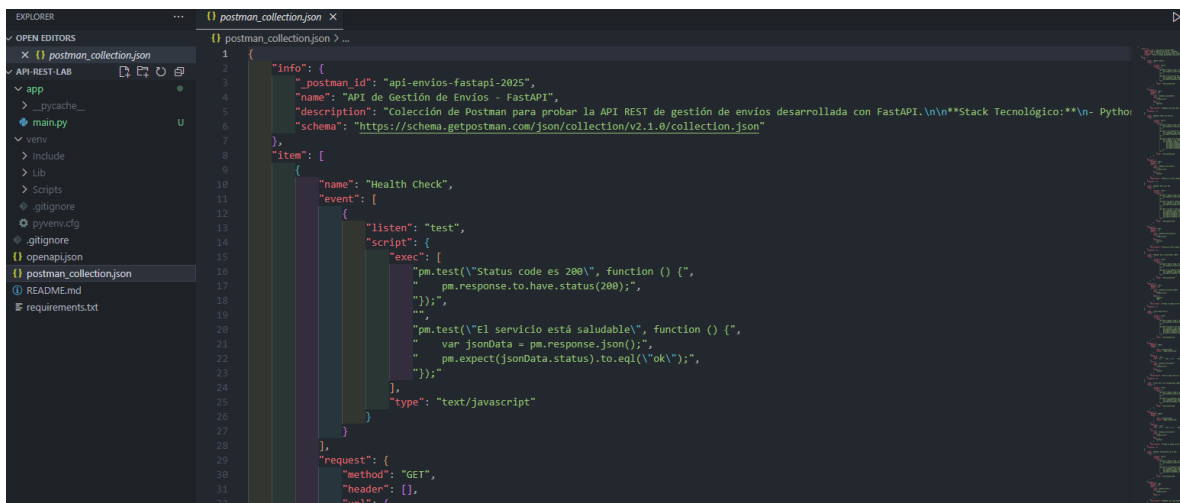
- URL base: `http://localhost:8000`.
- Método HTTP correspondiente (GET o POST).
- En el caso de POST /envios, se utilizó un cuerpo JSON con los campos del modelo Envío.

Después de verificar que todas las peticiones devolvían las respuestas esperadas, la colección se exportó en formato `.json`, cumpliendo el requisito de entregar una colección de pruebas con evidencias de cada endpoint.

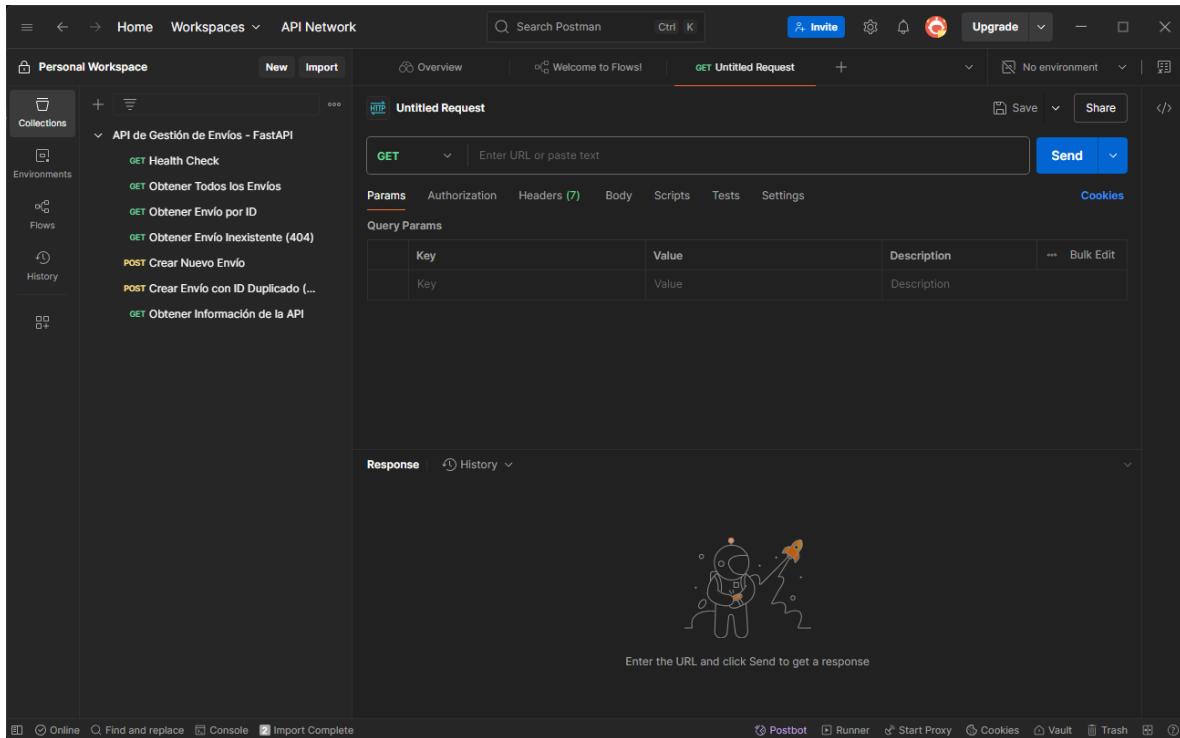
Importamos la colección a Postman:



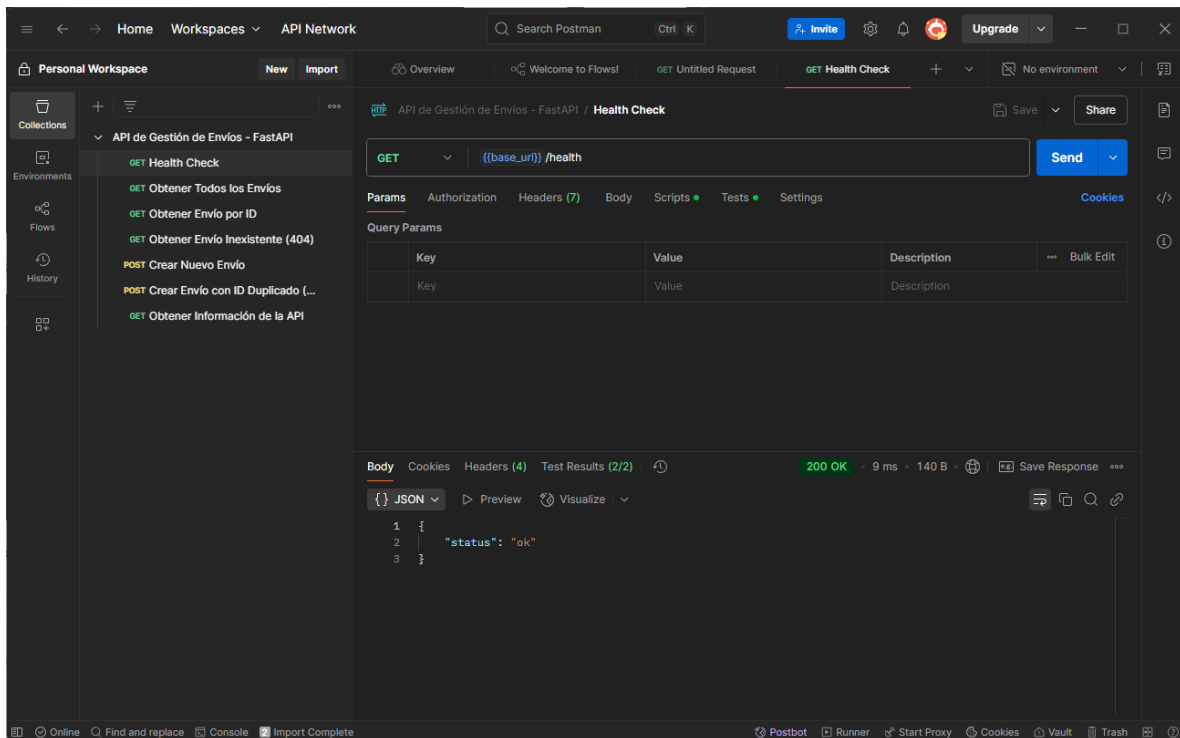
Este es el archivo .json que importaremos postman_collection.json:



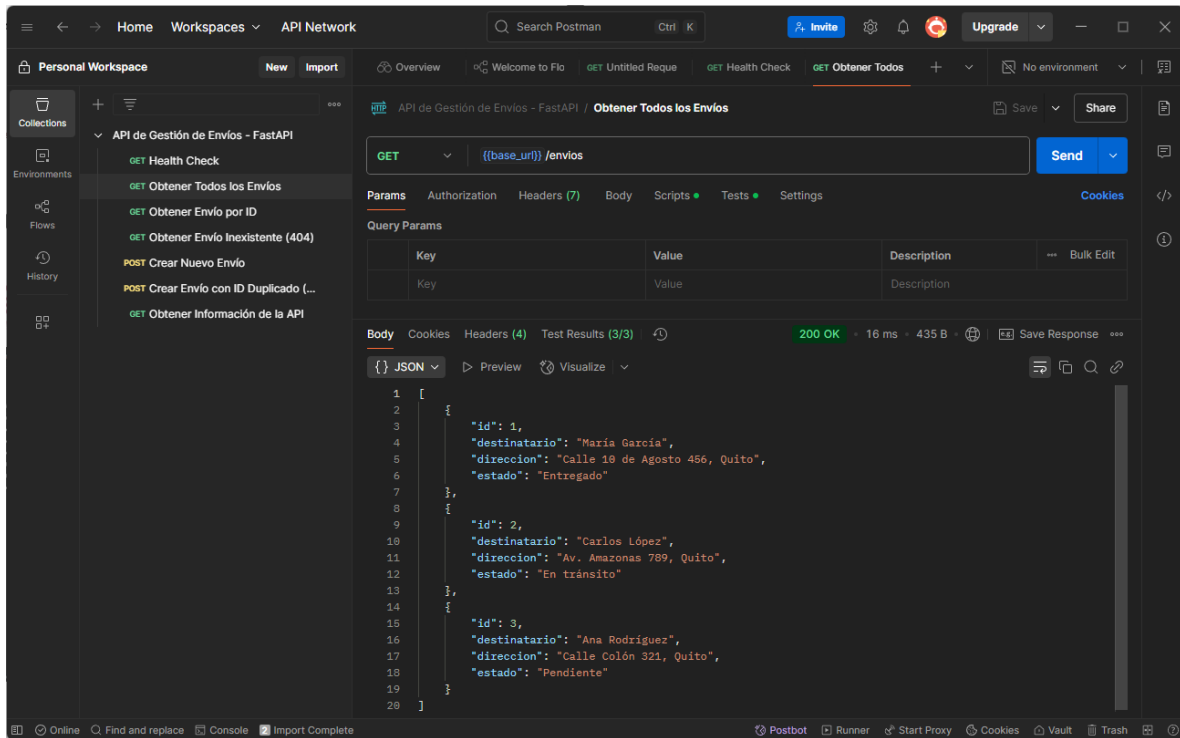
Se importó correctamente:



Ahora, realizaremos todas las pruebas de endpoints en Postman:
Health



Obtener envíos:



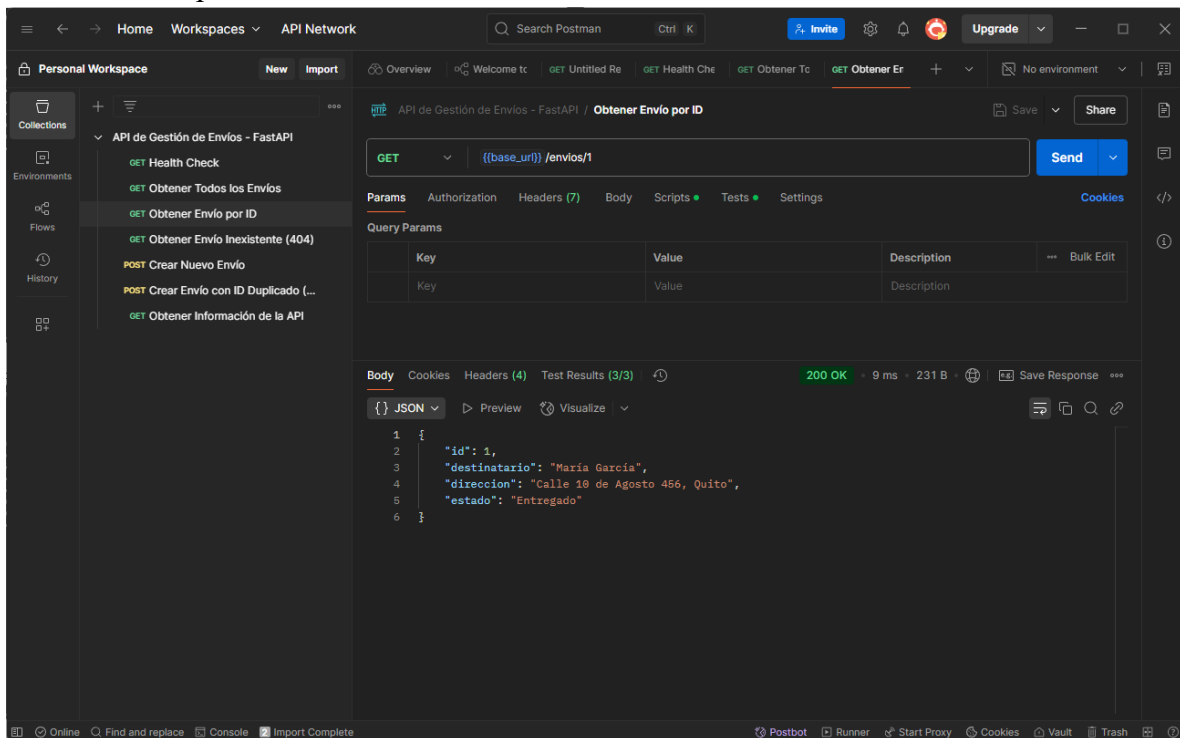
The screenshot shows the Postman interface for a collection named "API de Gestión de Envíos - FastAPI". The selected request is "Obtener Todos los Envíos" (GET /envios). The response is a 200 OK status with a JSON body containing an array of 3 shipment objects.

Key	Value	Description
id	1	
destinatario	"Maria Garcia"	
direccion	"Calle 10 de Agosto 456, Quito"	
estado	"Entregado"	

Key	Value	Description
id	2	
destinatario	"Carlos López"	
direccion	"Av. Amazonas 789, Quito"	
estado	"En tránsito"	

Key	Value	Description
id	3	
destinatario	"Ana Rodriguez"	
direccion	"Calle Colón 321, Quito"	
estado	"Pendiente"	

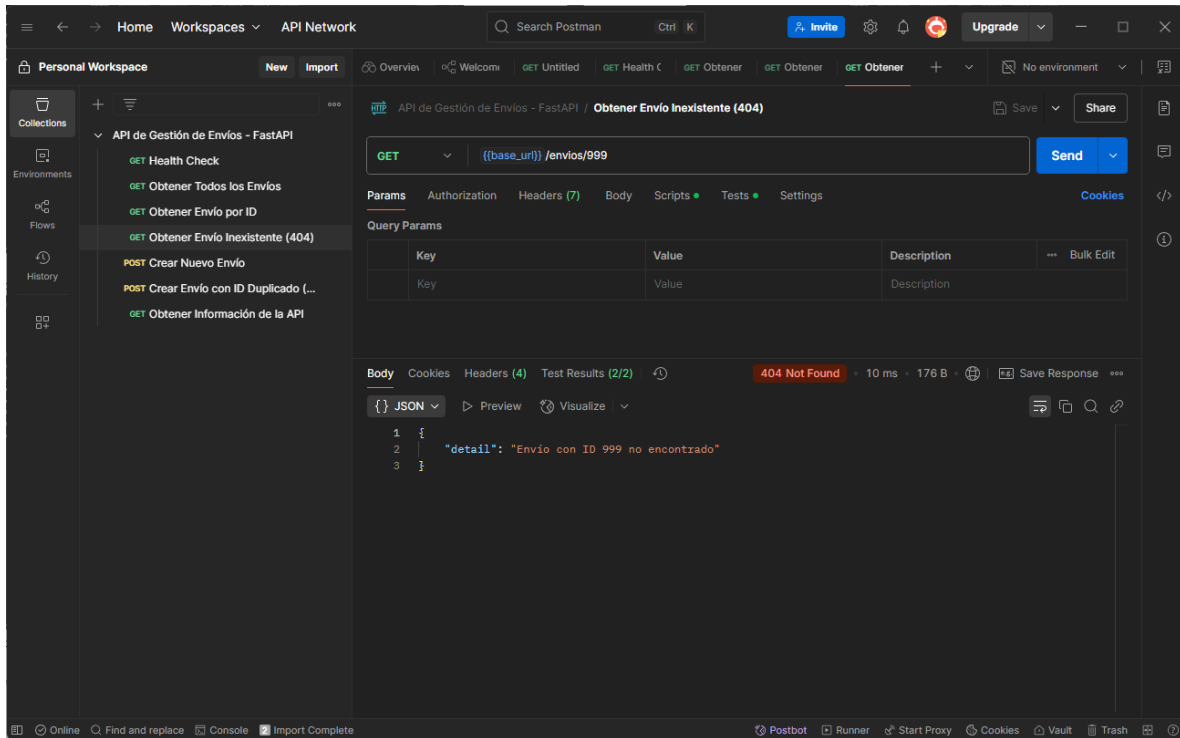
Obtener envío por ID:



The screenshot shows the Postman interface for the same collection. The selected request is "Obtener Envío por ID" (GET /envios/1). The response is a 200 OK status with a JSON body containing a single shipment object.

Key	Value	Description
id	1	
destinatario	"Maria Garcia"	
direccion	"Calle 10 de Agosto 456, Quito"	
estado	"Entregado"	

Obtener envío inexistente:

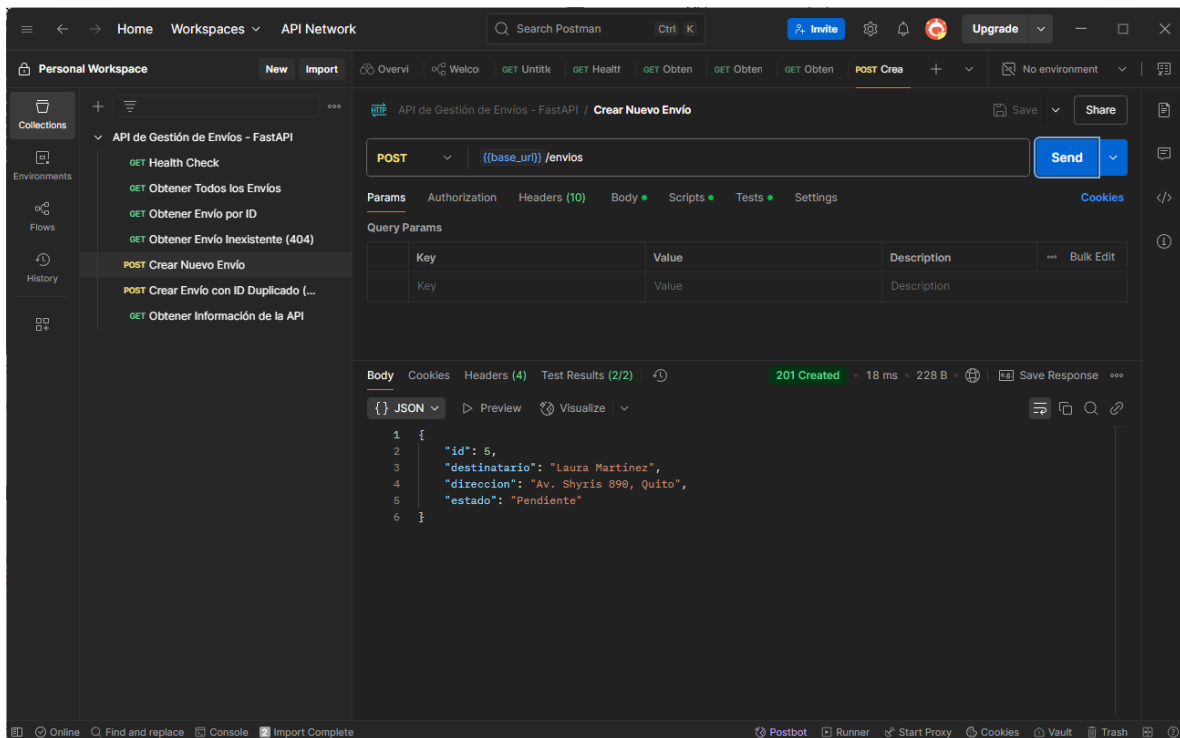


The screenshot shows the Postman interface with a GET request to `{{base_url}}/envios/999`. The response is a 404 Not Found status with a JSON body:

```
{  "detail": "Envío con ID 999 no encontrado"}
```

The interface includes a sidebar with collections, a top navigation bar, and a main workspace area with tabs for Params, Headers, Body, and Test Results.

Crear nuevo envío:

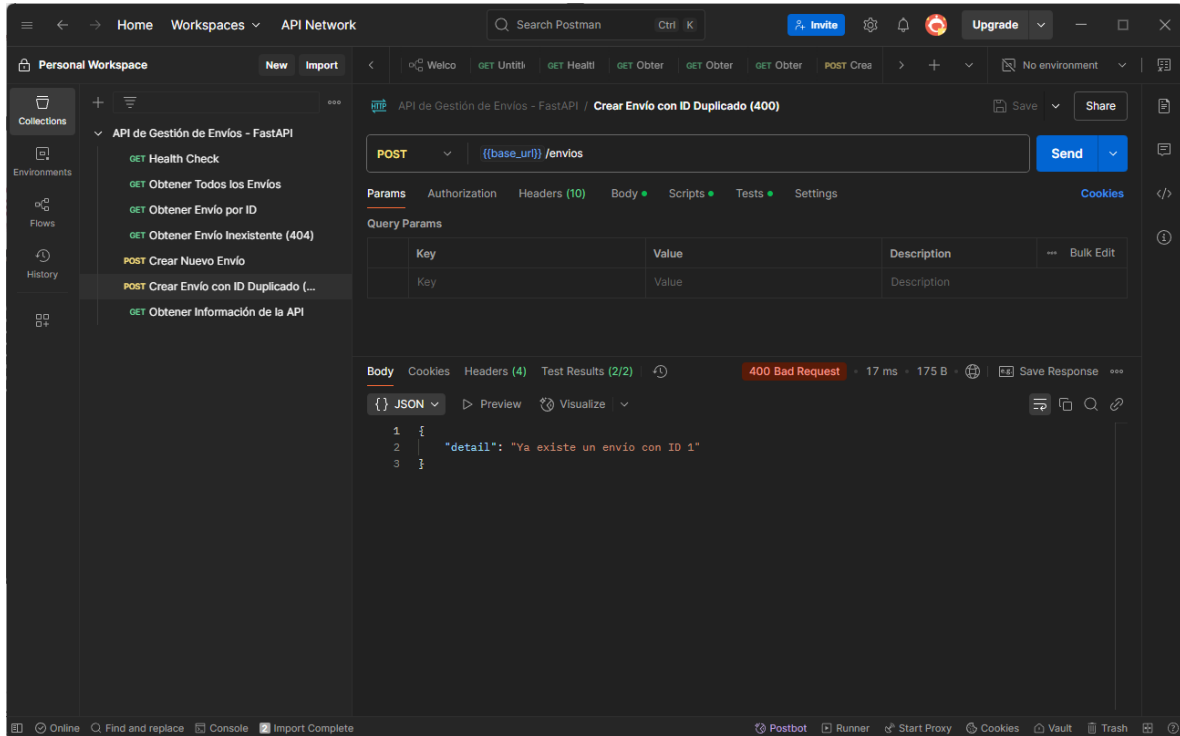


The screenshot shows the Postman interface with a POST request to `{{base_url}}/envios`. The response is a 201 Created status with a JSON body:

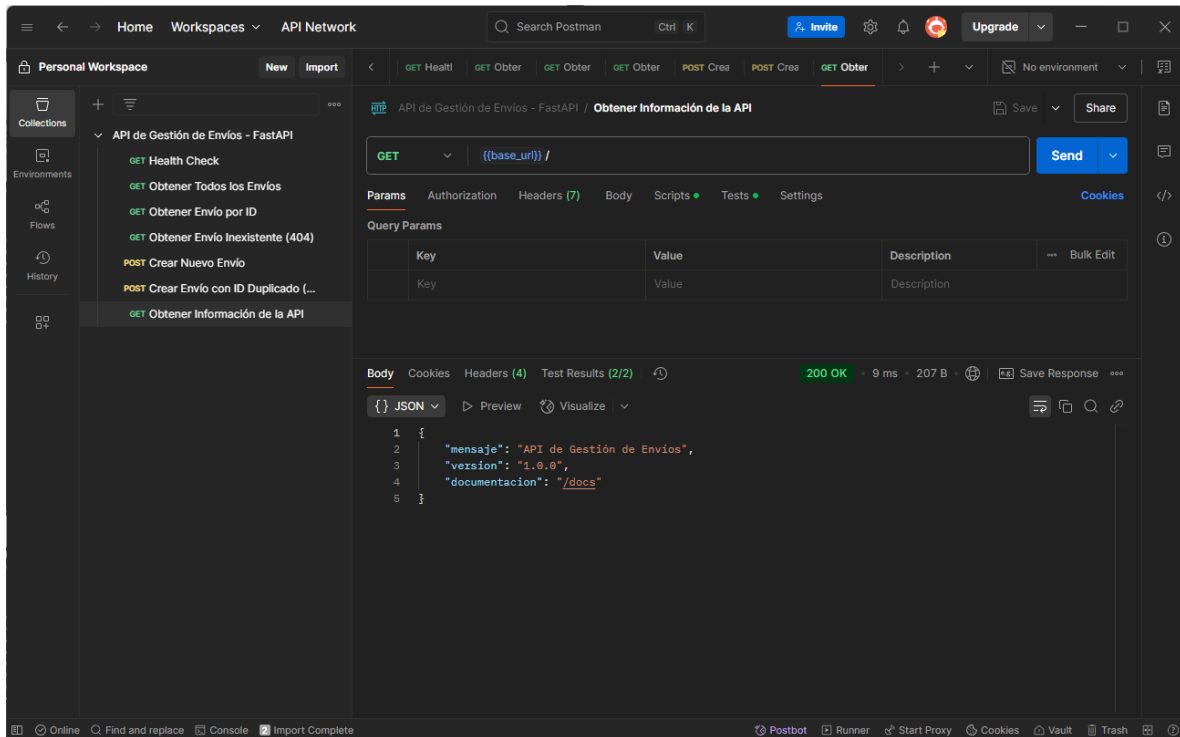
```
{  "id": 5,  "destinatario": "Laura Martinez",  "direccion": "Av. Shyris 898, Quito",  "estado": "Pendiente"}
```

The interface includes a sidebar with collections, a top navigation bar, and a main workspace area with tabs for Params, Headers, Body, and Test Results.

Crear envío con ID duplicado:



Obtener información de la API:



2.9. Organización de evidencias y documentación

Finalmente, se organizaron todos los artefactos del proyecto conforme a los entregables establecidos en el taller:

- Código fuente de la API (main.py).
- Contrato OpenAPI (openapi.json).
- Colección de Postman exportada en formato JSON.
- README técnico con pasos de instalación, ejecución y pruebas.
- Capturas de pantalla evidenciando el funcionamiento de cada endpoint.
- Informe escrito con la descripción detallada del proceso, conclusiones y recomendaciones.

Esta organización facilita la revisión por parte del tutor, asegura la trazabilidad del desarrollo y demuestra el cumplimiento de los criterios de evaluación definidos.

3. Link a repositorio GIT

<https://github.com/daniel-vizcarra/api-rest-lab.git>

4. Preguntas

1. ¿Qué ventajas ofrece una API REST bien diseñada frente a los enfoques tradicionales de integración?

Una API REST bien diseñada presenta múltiples ventajas frente a métodos tradicionales como la transferencia de archivos o la integración punto a punto. En primer lugar, REST permite interoperabilidad entre sistemas heterogéneos utilizando estándares ampliamente adoptados como HTTP y JSON. Además, favorece el bajo acoplamiento, de manera que los sistemas pueden evolucionar de forma independiente sin afectar a los consumidores del servicio.

Otra ventaja relevante es la escalabilidad, ya que REST se adapta fácilmente a arquitecturas distribuidas y entornos en la nube. La claridad en la definición de recursos y rutas facilita el mantenimiento, la documentación y la incorporación de nuevos desarrolladores. Finalmente, herramientas como OpenAPI permiten documentar la API de forma estandarizada, generando contratos claros y posibilitando la automatización de pruebas y clientes. Esto incrementa la confiabilidad y reduce errores de integración.

2. ¿Qué patrón de integración se aplicó en esta solución?

El patrón principal aplicado en esta solución es el Patrón de Servicio (Service API), el cual consiste en exponer funcionalidades como servicios accesibles mediante interfaces bien

definidas. Este patrón favorece la integración modular entre sistemas, permitiendo que los consumidores accedan a recursos como “envíos” a través de endpoints estandarizados siguiendo los principios RESTful.

Adicionalmente, se hace referencia al patrón de Request-Reply, ya que cada operación ejecutada por el cliente recibe una respuesta inmediata del servidor en formato JSON. Este tipo de interacción permite consultas directas, operaciones CRUD y validaciones inmediatas, lo cual resulta apropiado para escenarios modernos de integración basados en APIs.

3. ¿Cuál es la importancia del contrato OpenAPI dentro del desarrollo de la API?

El contrato OpenAPI desempeña un rol central en el diseño y la integración de la API. Su importancia radica en que es la fuente de verdad que describe de forma estandarizada los endpoints, parámetros, cuerpos de petición, tipos de respuesta, códigos de estado y modelos de datos. Esto garantiza que todos los desarrolladores y sistemas externos consuman la API de manera consistente.

Además, el contrato facilita la documentación automática, la generación de clientes en distintos lenguajes, la validación de solicitudes, la detección anticipada de errores y la automatización de pruebas. Desde una perspectiva de integración empresarial, disponer de un contrato claro reduce los riesgos, evita malentendidos y acelera el proceso de adopción de la API por parte de otras aplicaciones.

4. ¿Por qué es necesario exponer un endpoint de salud (/health)?

El endpoint `/health` permite verificar rápidamente si el servicio está activo y funcionando correctamente sin necesidad de ejecutar las operaciones completas del sistema. Este endpoint es especialmente importante en entornos de producción, despliegues automatizados, contenedores Docker o sistemas de monitoreo, donde herramientas externas requieren verificar el estado del servicio de forma periódica.

La presencia de un endpoint de health check asegura:

- Monitoreo continuo del estado del servicio,
- Detección anticipada de fallas,
- Facilidad para sistemas de orquestación (como Kubernetes),
- Mayor confiabilidad y disponibilidad del API.

Su simplicidad lo convierte en un componente esencial dentro de cualquier arquitectura moderna basada en servicios.

5. ¿Cuál es la relación entre las pruebas en Postman y la calidad de la integración?

Las pruebas realizadas en Postman permiten validar que cada endpoint responde de acuerdo con lo esperado, siguiendo los parámetros definidos en el contrato OpenAPI. Estas pruebas aseguran que la API es funcional, consistente y confiable, lo cual es indispensable para una integración exitosa.

Postman también facilita la creación de colecciones que pueden ser reutilizadas por otros equipos, garantizando la repetibilidad de las pruebas, reduciendo errores humanos y brindando una forma estandarizada de validar la API antes de ser consumida en entornos reales. Al exportar la colección, se proporciona una evidencia clara de que la API cumple con los requisitos funcionales establecidos.

6. ¿Qué mejoras podrían implementarse sobre la solución actual?

Aunque la implementación es funcional y cumple con los requisitos del taller, se identifican varias oportunidades de mejora:

- Incorporar una base de datos real para persistir los envíos.
- Implementar validaciones más estrictas en el modelo Envío.
- Integrar un sistema de logs estructurados para facilitar auditoría.
- Añadir pruebas automatizadas con pytest o herramientas similares.
- Manejar errores de forma más robusta utilizando excepciones personalizadas.
- Implementar paginación en el endpoint GET /envios.

Estas mejoras permitirían escalar la API hacia entornos empresariales reales y elevar el nivel de robustez de la solución.

5. Conclusiones y Recomendaciones

Conclusiones

- La implementación de la API REST con FastAPI permitió comprender de manera práctica los principios fundamentales de una arquitectura orientada a recursos. La estructura clara de los endpoints, combinada con la validación automática del modelo Envío, facilitó la creación de un servicio consistente y alineado con los estándares REST. Además, la generación automática del contrato OpenAPI evidenció la importancia de mantener una documentación precisa que facilite la interoperabilidad entre sistemas.
- El uso del stack Python + FastAPI demostró ser una alternativa eficiente frente a enfoques tradicionales de integración. La disponibilidad inmediata de Swagger y la simplicidad para ejecutar el servicio permitieron acelerar el proceso de desarrollo, sin

sacrificar calidad técnica. Las pruebas realizadas mediante Postman confirmaron la correcta funcionalidad de la API y reforzaron la importancia de validar cada operación para asegurar un comportamiento confiable y reproducible.

Recomendaciones

- Se recomienda evolucionar la API incorporando persistencia mediante una base de datos relacional o NoSQL, con el fin de almacenar los envíos de manera permanente. Asimismo, sería conveniente incluir un sistema robusto de manejo de errores y respuestas estándar, lo cual mejoraría la experiencia de los consumidores del servicio y alinearía la API con prácticas profesionales de desarrollo backend.
- Como mejora adicional, se sugiere integrar herramientas de monitoreo y registros estructurados que permitan evaluar el rendimiento y detectar fallos en tiempo real. También sería valioso complementar el proyecto con pruebas automatizadas para garantizar la estabilidad del servicio ante futuros cambios de alcance o incrementos en la complejidad funcional.