

**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:

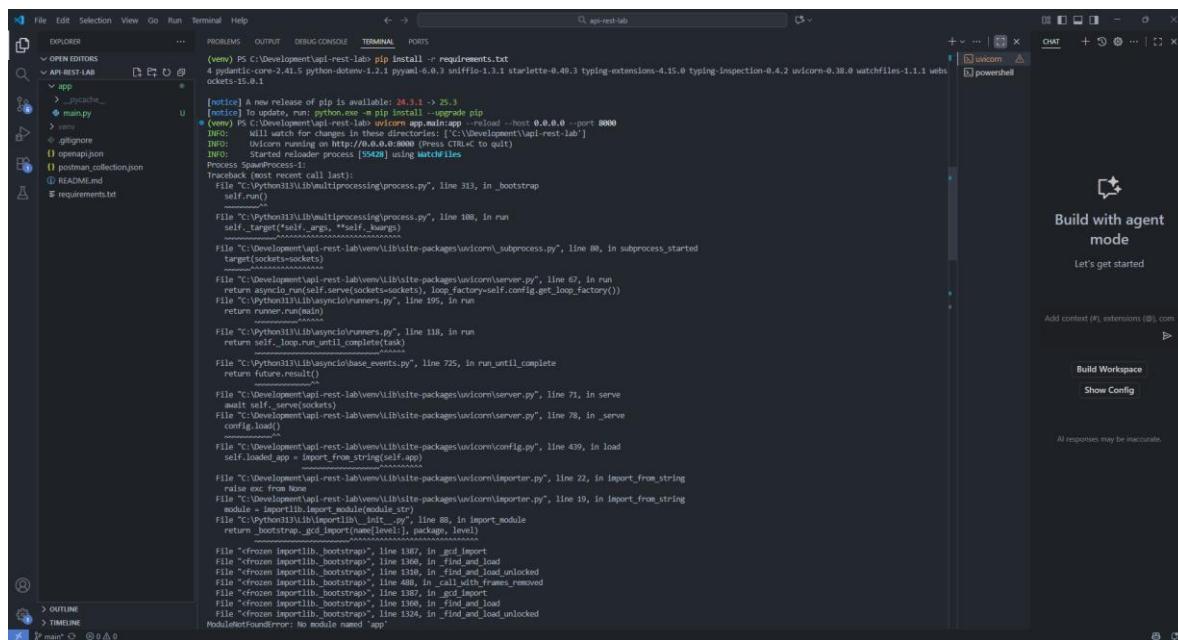
- FastAPI ofrece soporte nativo para **OpenAPI 3.0** y genera automáticamente la documentación Swagger.
- Permite implementar APIs REST de forma rápida, manteniendo buenas prácticas de diseño.
- La ejecución del servicio es reproducible mediante un solo comando (uvicorn), cumpliendo con las condiciones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

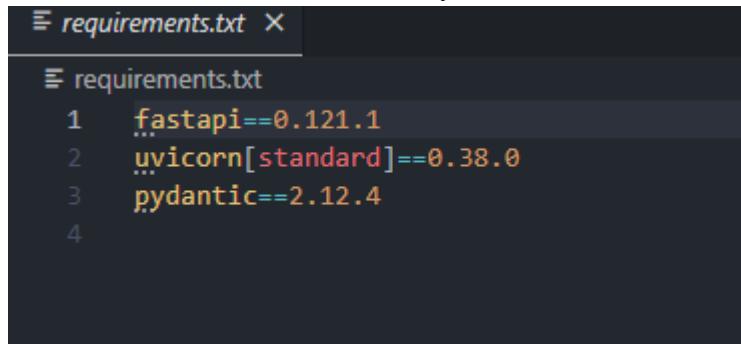
Con esto, el entorno quedó listo para iniciar la implementación de la API.

```
PS C:\Development\api-rest-lab> python -m venv venv
PS C:\Development\api-rest-lab> venv\Scripts\activate
(venv) PS C:\Development\api-rest-lab> pip install -r requirements.txt
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 -r requirements.txt (line 2))
  Downloading uvicorn-0.38.0-(py3|uvicorn|standard)-none-any.whl.metadata (25 kB)
Collecting pydantic==2.12.4 (from -r requirements.txt (line 3))
  Downloading pydantic-2.12.4-py3-none-any.whl.metadata (89 kB)
Collecting pydantic-core==2.41.5 (from -r requirements.txt (line 4))
  Downloading pydantic-core-2.41.5-cp313-cp313-win_amd64.whl.metadata (6.4 kB)
Collecting typing_extensions==4.0.0 (from fastapi==0.121.1->r requirements.txt (line 5))
  Downloading typing_extensions-4.0.0-py3-none-any.whl.metadata (3.3 kB)
Collecting httpcore==0.2.1 (from -r requirements.txt (line 6))
  Downloading httpcore-0.2.1-py3-none-any.whl.metadata (6.6 kB)
Collecting annotated_type==0.0.4-py3-none-any.whl.metadata (6.6 kB)
Collecting click==7.0 (from uvicorn==0.38.0->uvicorn[standard]==0.38.0->r requirements.txt (line 2))
  Downloading click-7.0-py3-none-any.whl.metadata (2.8 kB)
Collecting httpx==0.22.0 (from -r requirements.txt (line 7))
  Downloading httpx-0.22.0-py3-none-any.whl.metadata (8.3 kB)
Collecting annotated_type==0.0.6 (from pydantic==2.12.4->r requirements.txt (line 3))
  Downloading annotated_type-0.0.6-py3-none-any.whl.metadata (19 kB)
Collecting httpcore==0.2.1 (from -r requirements.txt (line 6))
  Downloading httpcore-0.2.1-py3-none-any.whl.metadata (6.6 kB)
Collecting typing_inspect==0.4.2 (from pydantic==2.12.4->r requirements.txt (line 3))
  Downloading typing_inspect-0.4.2-py3-none-any.whl.metadata (2.6 kB)
Collecting typeguard==3.8.1 (from -r requirements.txt (line 8))
  Downloading typeguard-3.8.1-py3-none-any.whl.metadata (39.3 kB)
Using cached colorama-0.4.0-py2.py3-none-any.whl.metadata (17 kB)
Collecting httpx==0.22.0 (from uvicorn[standard]==0.38.0->r requirements.txt (line 2))
  Downloading httpx-0.22.0-py3-none-any.whl.metadata (8.3 kB)
Collecting httpcore==0.2.1 (from -r requirements.txt (line 6))
  Downloading httpcore-0.2.1-py3-none-any.whl.metadata (6.6 kB)
Collecting pydantic==2.12.1-py3-none-any.whl.metadata (25 kB)
Collecting pydantic==2.12.1 (from uvicorn[standard]==0.38.0->r requirements.txt (line 2))
  Downloading pydantic-2.12.1-py3-none-any.whl.metadata (89 kB)
Collecting uvicorn==0.38.0 (from uvicorn[standard]==0.38.0->r requirements.txt (line 2))
  Downloading uvicorn-0.38.0-(py3|uvicorn|standard)-none-any.whl.metadata (25 kB)
Collecting websockets==10.1.1 (from uvicorn[standard]==0.38.0->r requirements.txt (line 2))
  Downloading websockets-10.1.1-cp313-cp313-win_amd64.whl.metadata (5.0 kB)
Collecting httpcore==0.2.1 (from -r requirements.txt (line 6))
  Downloading httpcore-0.2.1-py3-none-any.whl.metadata (6.6 kB)
Collecting asyncio==3.6.3 (from starlette==0.10.0->fastapi==0.121.1->r requirements.txt (line 1))
  Downloading asyncio-3.6.3-py3-none-any.whl.metadata (4.1 kB)
Collecting idna==2.1 (from asyncio==3.6.3->starlette==0.10.0->fastapi==0.121.1->r requirements.txt (line 1))
  Downloading idna-2.1-py3-none-any.whl.metadata (10 kB)
Collecting sniffio==1.1.1 (from asyncio==3.6.3->starlette==0.10.0->fastapi==0.121.1->r requirements.txt (line 1))
  Downloading sniffio-1.1.1-py3-none-any.whl.metadata (10 kB)
Collecting pydantic==2.12.4-py3-none-any.whl (40 kB)
Downloaded pydantic_core-2.41.5-cp313-cp313-win_amd64.whl (2.0 kB)
2.07.0 MB 39.0 MB/s eta 0:00:00
Collecting annotated_type==0.0.4-py3-none-any.whl (13 kB)
Downloaded annotated_type-0.0.4-py3-none-any.whl (13 kB)
Collecting click==3.0-py3-none-any.whl (187 kB)
Using cached colorama-0.4.0-py2.py3-none-any.whl (25 kB)
Downloaded httpcore-0.2.1-py3-none-any.whl (85 kB)
Downloaded httpx-0.22.0-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-dates==1.2.1 pyyaml==6.0.2 uvicore==1.3.1 starlette==0.49.3 typing-extensions==4.15.0 typing-inspection==0.4.2 uvicorn==0.38.0 watchfiles==1.1.1 websockets==15.0.1
[...]
```

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
```

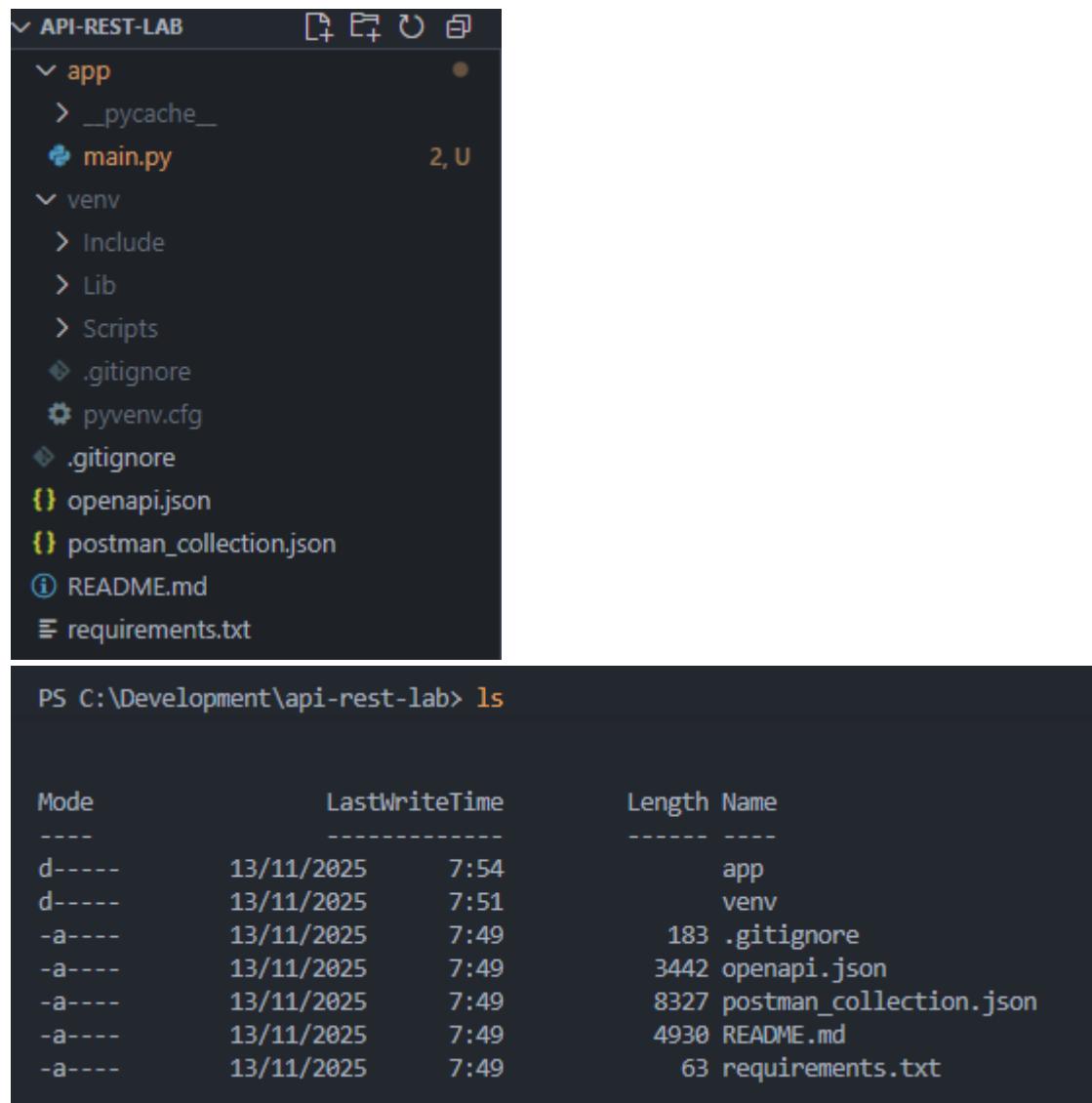
Con esto ya tenemos uvicorn ara realizar las pruebas.

## 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.



The screenshot shows a file explorer window titled "API-REST-LAB" and a terminal window below it.

**File Explorer Content:**

- app
- \_\_pycache\_\_
- main.py
- venv
- Include
- Lib
- Scripts
- .gitignore
- pyvenv.cfg
- .gitignore
- openapi.json
- postman\_collection.json
- README.md
- requirements.txt

**Terminal Content:**

```
PS C:\Development\api-rest-lab> ls
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	13/11/2025	7:54	app
d----	13/11/2025	7:51	venv
-a---	13/11/2025	7:49	183 .gitignore
-a---	13/11/2025	7:49	3442 openapi.json
-a---	13/11/2025	7:49	8327 postman_collection.json
-a---	13/11/2025	7:49	4930 README.md
-a---	13/11/2025	7:49	63 requirements.txt

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 Envio:** contiene atributos como id, destinatario, dirección 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", dirección="Calle 10 de Agosto 456, Quito", estado="Entregado"),
    Envio(id=2, destinatario="Carlos López", dirección="Av. Amazonas 789, Quito", estado="En tránsito"),
    Envio(id=3, destinatario="Ana Rodríguez", dirección="Calle Colón 321, Quito", estado="Pendiente")
]
```

Definición de la clase Envio con sus campos (id, destinatario, dirección, 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 {env.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.



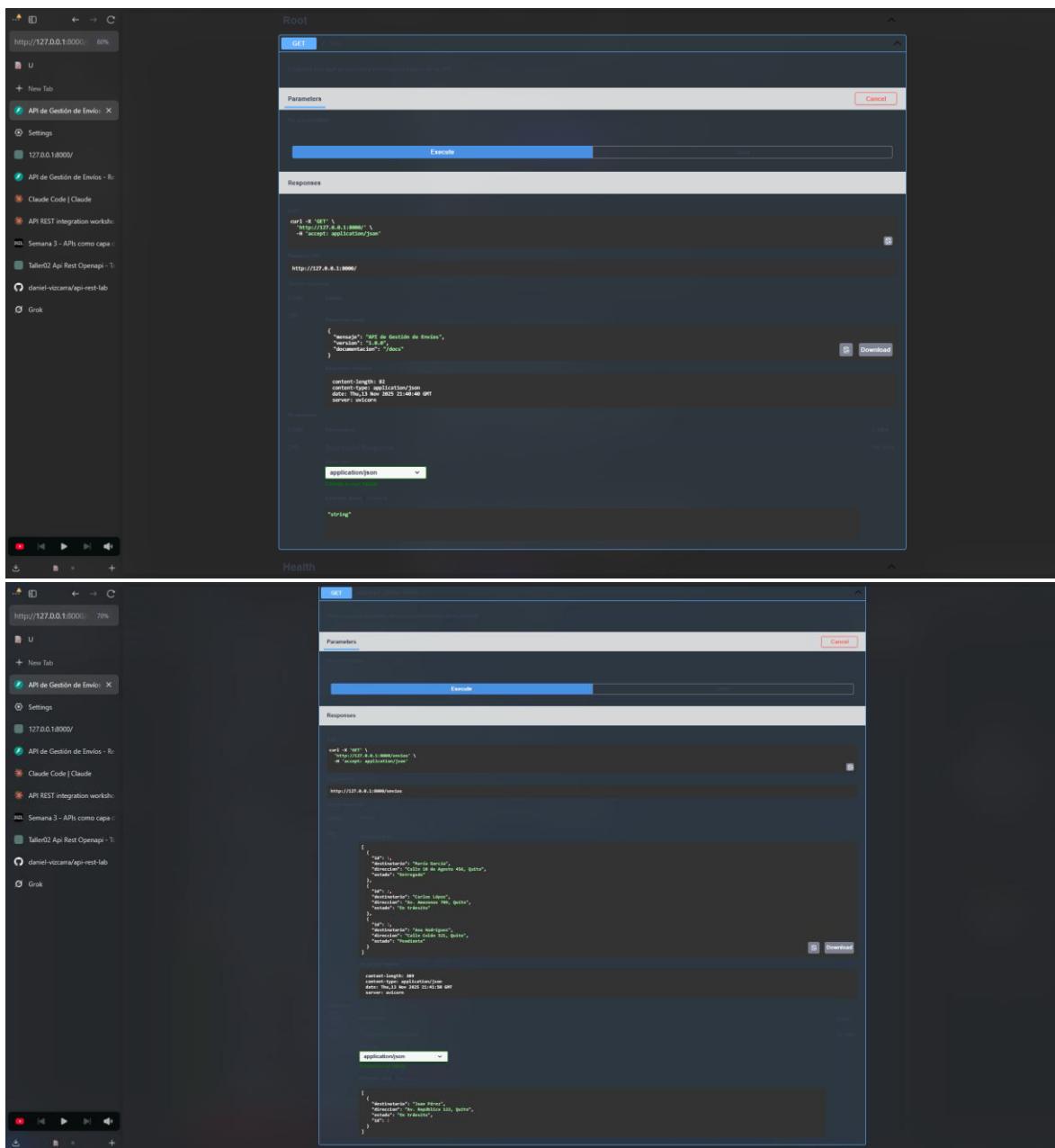
The screenshot shows a Microsoft Visual Studio Code interface with a terminal window open. The terminal output is as follows:

```
PS C:\Development\api-rest-lab> uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
File "C:\Development\api-rest-lab\venv\lib\site-packages\uvicorn\server.py", line 67, in run
    return asyncio.run(self._server.sockets(sockets), loop=loop, config=config.get_loop_factory())
File "C:\Python311\lib\asyncio\runners.py", line 195, in run
    return runner.run()
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._server.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_name)
File "C:\Python311\lib\importlib\__init__.py", line 98, in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
File "C:\Python311\lib\importlib\bootstrap.py", line 380, in _gcd_import
    _find_and_load(bootstrap, __name__, __spec__, wrap)
File "C:\Python311\lib\importlib\bootstrap.py", line 350, in _find_and_load
    _find_and_load_unlocked(__name__)
File "C:\Python311\lib\importlib\bootstrap.py", line 430, in _call_with_frames_removed
    _find_and_load_unlocked(__name__)
File "C:\Python311\lib\importlib\bootstrap.py", line 350, in _find_and_load
    _find_and_load_unlocked(__name__)
File "C:\Python311\lib\importlib\bootstrap.py", line 1324, in _find_and_load_unlocked
ModuleNotFoundError: No module named 'app'
WARNING: [main.py] detected changes in "main.py", "app/main.py". Reloading...
INFO: Started server process [35428]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:8000 - "-" [19/Oct/2023:10:45:11 +0000] "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:53627 - "-" [19/Oct/2023:10:45:11 +0000] "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:53627 - "-" [19/Oct/2023:10:45:11 +0000] "GET /HTTP/1.1" 200 OK
INFO: 127.0.0.1:53749 - "-" [19/Oct/2023:10:45:11 +0000] "GET /docs/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:53749 - "-" [19/Oct/2023:10:45:11 +0000] "GET /docs/_/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:53756 - "-" [19/Oct/2023:10:45:11 +0000] "GET /redoc/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:53756 - "-" [19/Oct/2023:10:45:11 +0000] "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:53778 - "-" [19/Oct/2023:10:45:11 +0000] "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:53778 - "-" [19/Oct/2023:10:45:11 +0000] "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: Shutting down
INFO: Waiting for application shutdown.
INFO: Application shutdown complete.
INFO: 127.0.0.1:53628 - "-" [19/Oct/2023:10:45:11 +0000] "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:53628 - "-" [19/Oct/2023:10:45:11 +0000] "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: PS C:\Development\api-rest-lab> curl http://127.0.0.1:8000/health
```

## Página principal de swagger

The screenshot shows the API Management interface for the "API de Gestión de Envíos" version 1.0.0. The left sidebar lists several open tabs, including "API de Gestión de Envío", "Settings", and "127.0.0.1:8000". The main content area displays the API's documentation with sections for "Root", "Health", "Envíos", and "Schemas". The "Root" section has a "GET" button. The "Health" section also has a "GET" button. The "Envíos" section contains three buttons: "GET" (highlighted in blue), "POST" (highlighted in green), and another "GET" button. The "Schemas" section shows three schema definitions: "Envio > `object`", "HTTPValidationError > `object`", and "ValidationError > `object`". The bottom of the screen features a toolbar with various icons.

Realizamos pruebas de diferentes endpoints con Swagger:



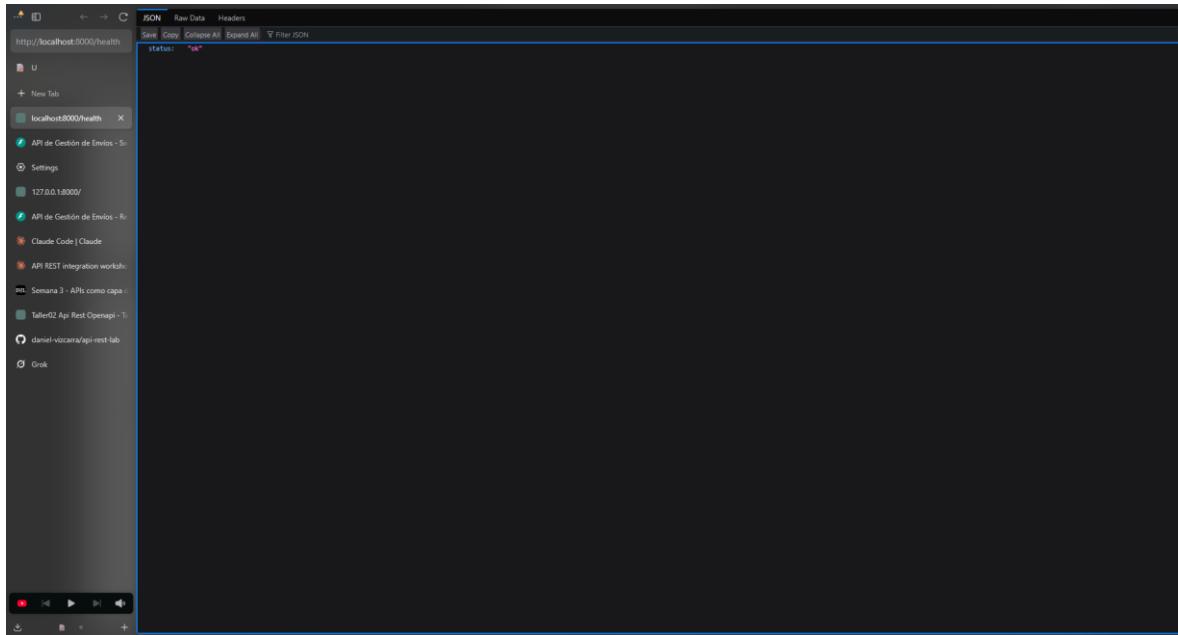
## 2.5. Endpoint de salud /health

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

**Root**

API de Gestión de Envíos (TDD)

Transient OpenAPI specification

API REST para gestionar envíos - Taller de Integración de Sistemas

Root

Responses

> 200 Successful Response

Health

Health Check

Endpoint usado para verificar que el servicio está funcionando correctamente. Útil para monitoreo y despliegues automatizados.

Responses

> 200 Successful Response

GET /

Response samples

200

Content type application/json

null

Copy

GET /health

Response samples

200

Content type application/json

null

Copy

## Health

**Health**

Endpoint usado para verificar que el servicio está funcionando correctamente. Útil para monitoreo y despliegues automatizados.

Responses

> 200 Successful Response

Envíos

Obtener Envíos

Endpoint de consulta para obtener los envíos almacenados en el sistema.

Responses

> 200 Successful Response

GET /envíos

Response samples

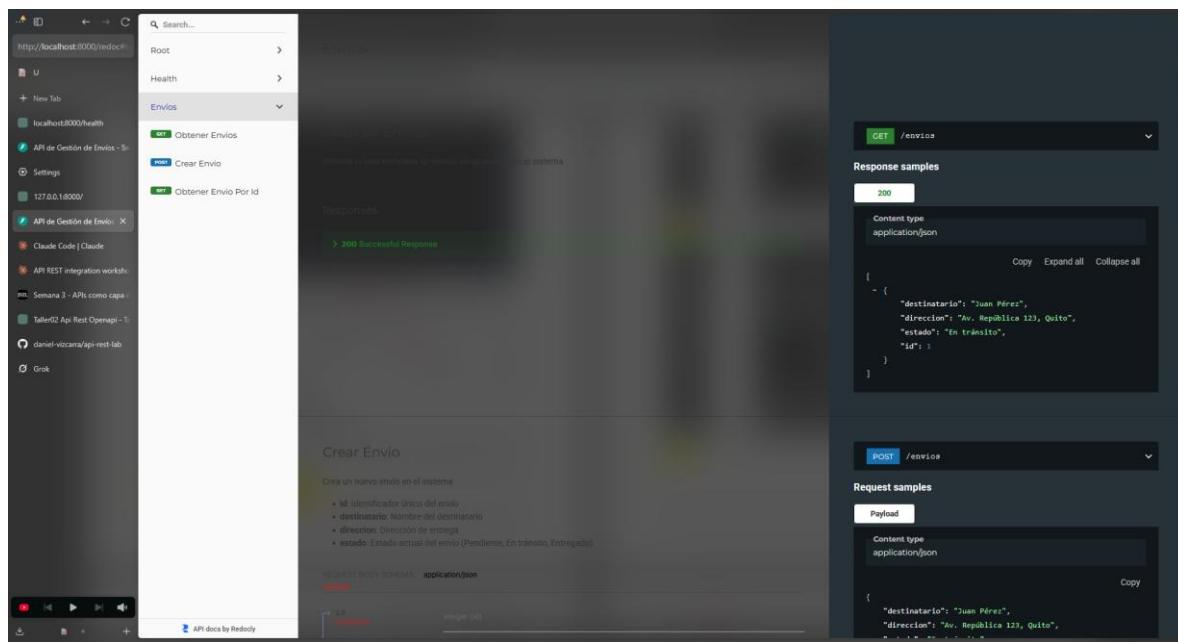
200

Content type application/json

[{"destinatario": "Juan Pérez", "dirección": "Av. República 123, Quito", "peso": 500, "origen": "Bogotá, Colombia"}]

Copy Expand all Collapse all

## 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> unicorn 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: Unicorn 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 Envio.

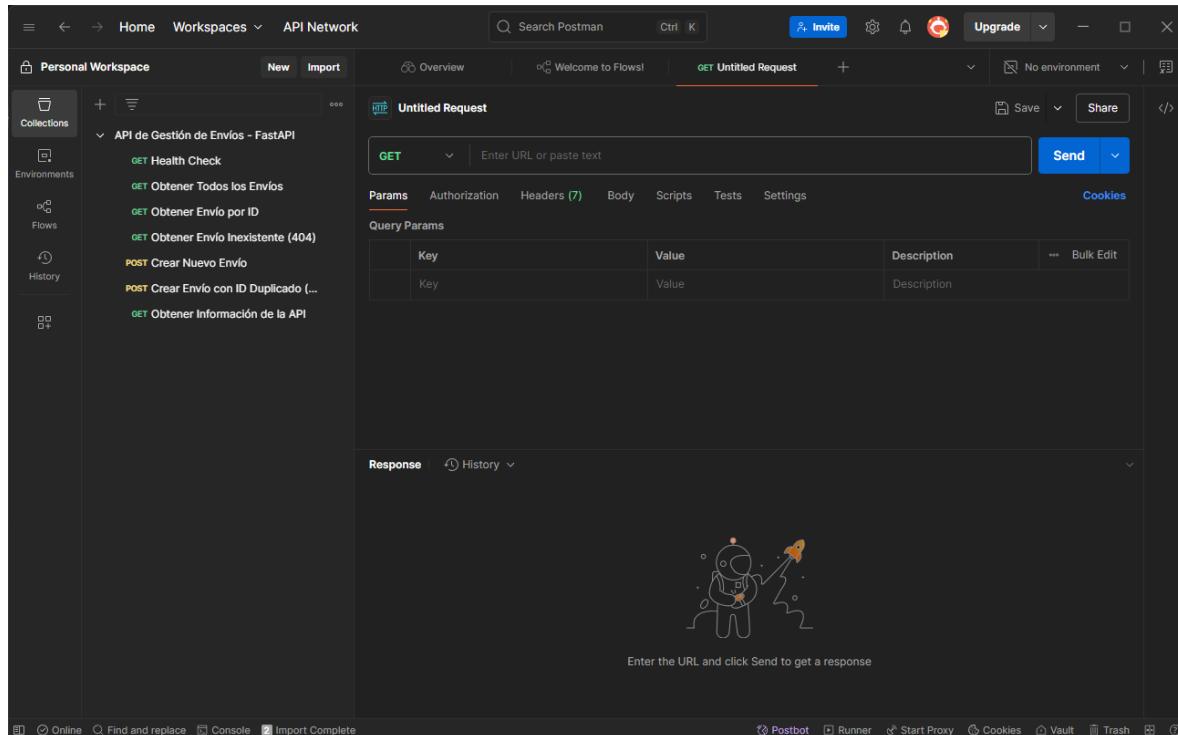
Después de verificar que todas las peticiones devolvían las respuestas esperadas, la colección se exportó en formato .json, cumpliendo el requisito de entregar una colección de pruebas con evidencias de cada endpoint.

Importamos la colección a Postman:

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'API Network', a search bar, and various icons for 'Invite', 'Upgrade', and settings. Below the navigation is a sidebar with sections for 'Personal Workspace', 'Collections', 'Environments', 'Flows', and 'History'. The main workspace is titled 'Untitled Request' and shows a 'GET' request with the URL field empty. A modal window is open, prompting the user to 'Paste cURL, Raw text or URL...' or to 'Drop anywhere to import' files or folders. The bottom of the screen features a toolbar with 'Create Collection', 'Migrate to Postman', 'Other Sources', 'Learn more about importing data', and a cartoon character icon. The status bar at the bottom includes links for 'Online', 'Find and replace', 'Console', 'Import Complete', 'Postbot', 'Runner', 'Start Proxy', 'Cookies', 'Vault', and 'Trash'.

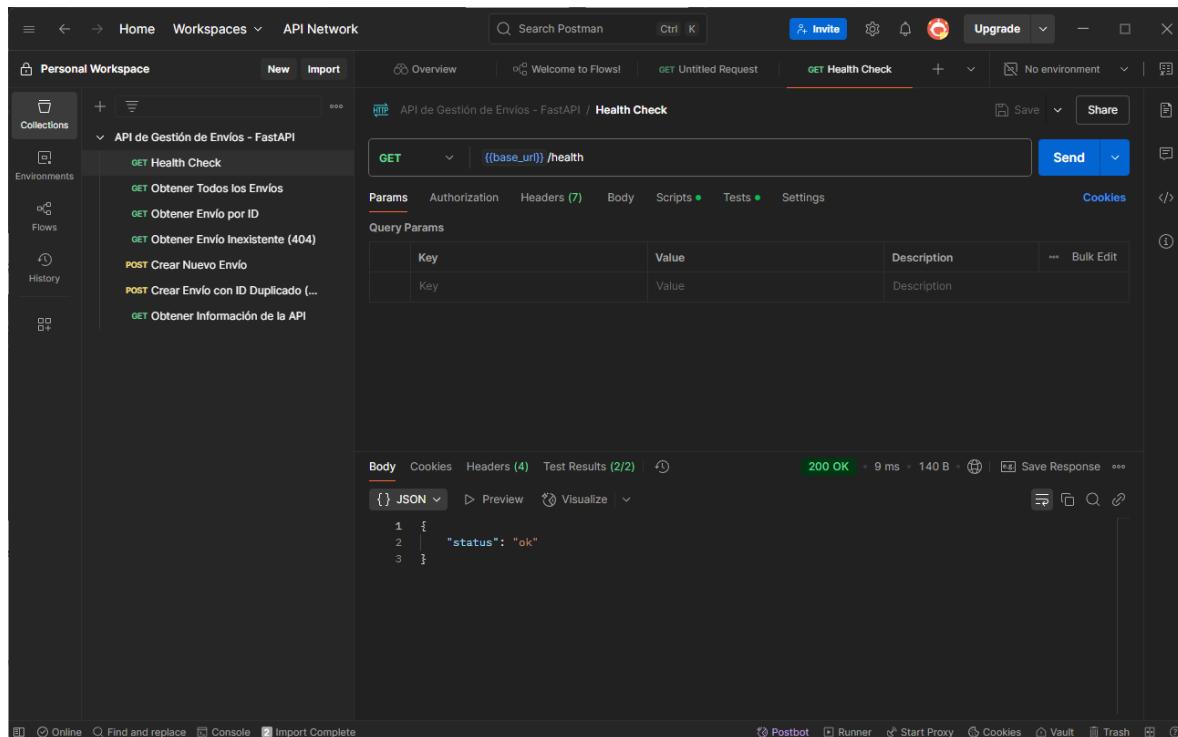
Este es el archivo .json que importaremos postman\_collection.json:

Se importó correctamente:



The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Personal Workspace' containing 'Collections', 'Environments', 'Flows', and 'History'. Below this is a '+' button and a 'New' button. The main area displays a collection named 'API de Gestión de Envíos - FastAPI' with several endpoints listed under it. One endpoint, 'GET Health Check', is currently selected. The top navigation bar includes 'Home', 'Workspaces', 'API Network', 'Overview', 'Welcome to Flows!', 'GET Untitled Request', 'Save', 'Share', 'Upgrade', and environment selection. A central panel shows an 'Untitled Request' with a 'GET' method and a placeholder URL 'Enter URL or paste text'. Below the request are tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Scripts', 'Tests', and 'Settings'. A 'Query Params' table is also present. At the bottom of the main panel, there's a 'Response' section with a cartoon character and the instruction 'Enter the URL and click Send to get a response'. The footer contains various status icons and links like 'Postbot', 'Runner', 'Start Proxy', 'Cookies', 'Vault', 'Trash', and help links.

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



This screenshot shows the same Postman interface as the previous one, but now the 'GET Health Check' endpoint is selected. The main panel displays the request details: 'GET {{base\_url}} /health'. The 'Params' tab is active, showing a single parameter 'Key' with the value 'Value'. The 'Headers' tab shows 'Content-Type: application/json'. The 'Body' tab is selected, showing a JSON response: { "status": "ok" }. The 'Cookies' and 'Headers' tabs are also visible. At the bottom, the response status is shown as '200 OK' with a duration of '9 ms' and a size of '140 B'. There are also 'Save Response' and other UI elements. The footer is identical to the first screenshot.

## Obtener envíos:

Personal Workspace

API Network

Overview Welcome to Flo GET Untitled Req GET Health Check GET Obtener Todos ...

GET /envios

Params Authorization Headers (7) Body Scripts Tests Settings Cookies

Key	Value	Description
Key	Value	Description

Body Cookies Headers (4) Test Results (3/3)

```

1 [
2   {
3     "id": 1,
4     "destinatario": "Maria Garcia",
5     "direccion": "Calle 10 de Agosto 456, Quito",
6     "estado": "Entregado"
7   },
8   {
9     "id": 2,
10    "destinatario": "Carlos Lopez",
11    "direccion": "Av. Amazonas 789, Quito",
12    "estado": "En tránsito"
13  },
14  {
15    "id": 3,
16    "destinatario": "Ana Rodriguez",
17    "direccion": "Calle Colón 321, Quito",
18    "estado": "Pendiente"
19  }
20 ]

```

200 OK 16 ms 435 B

## Obtener envío por ID:

Personal Workspace

API Network

Overview Welcome to Flo GET Untitled Req GET Health Check GET Obtener Envío por ID ...

GET /envios/1

Params Authorization Headers (7) Body Scripts Tests Settings Cookies

Key	Value	Description
Key	Value	Description

Body Cookies Headers (4) Test Results (3/3)

```

1 {
2   "id": 1,
3   "destinatario": "Maria Garcia",
4   "direccion": "Calle 10 de Agosto 456, Quito",
5   "estado": "Entregado"
6 }

```

200 OK 9 ms 231 B

## Obtener envío inexistente:

The screenshot shows the Postman interface with a collection named "API de Gestión de Envíos - FastAPI". A GET request is made to `http://{{base_url}}/envios/999`. The response status is 404 Not Found, with the message: "Envío con ID 999 no encontrado".

## Crear nuevo envío:

The screenshot shows the Postman interface with a collection named "API de Gestión de Envíos - FastAPI". A POST request is made to `http://{{base_url}}/envios` with the following JSON body:

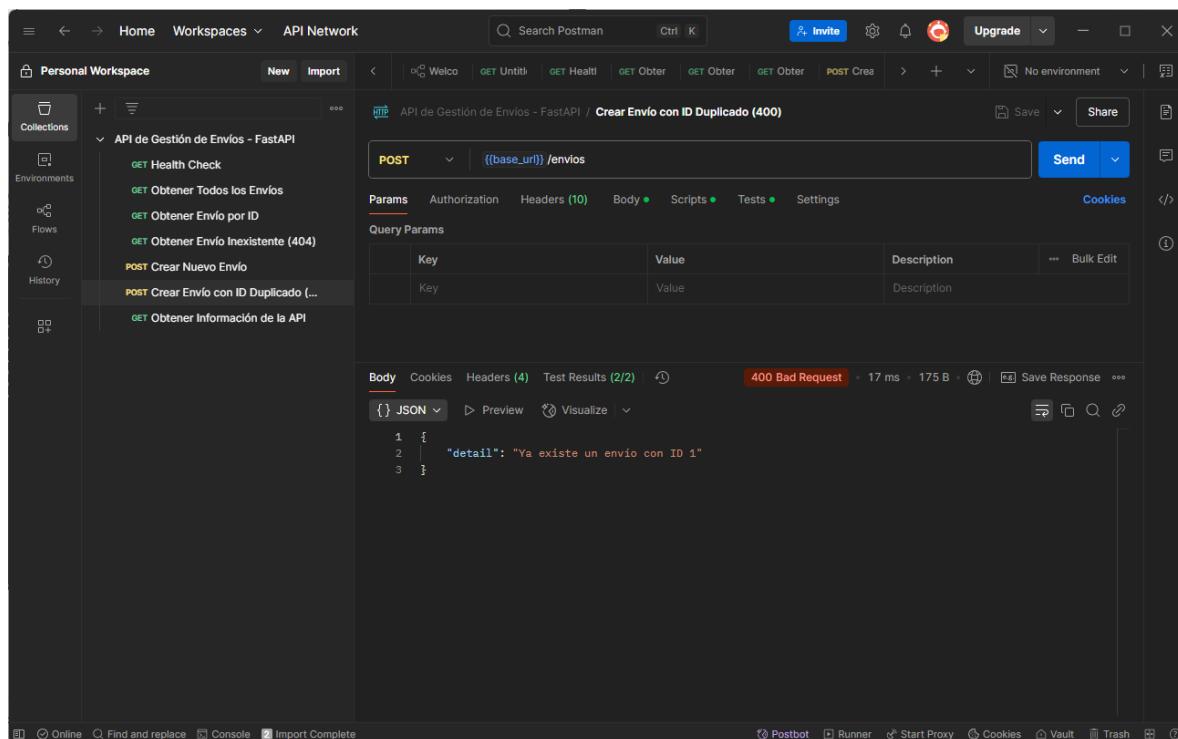
```

1 {
2   "id": 5,
3   "destinatario": "Laura Martínez",
4   "dirección": "Av. Shyris 890, Quito",
5   "estado": "Pendiente"
6 }

```

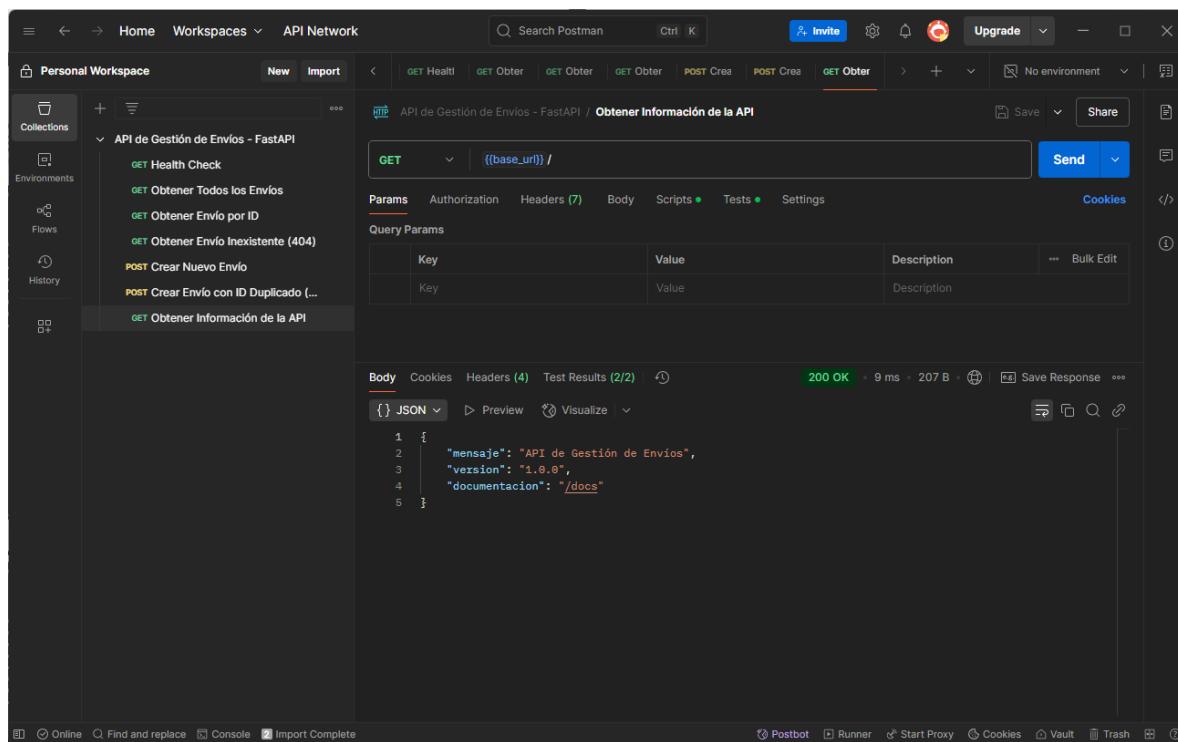
The response status is 201 Created.

## Crear envío con ID duplicado:



The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Personal Workspace' selected, showing a collection named 'API de Gestión de Envíos - FastAPI'. This collection contains several API endpoints: 'GET Health Check', 'GET Obtener Todos los Envíos', 'GET Obtener Envío por ID', 'GET Obtener Envío Inexistente (404)', 'POST Crear Nuevo Envío', 'POST Crear Envío con ID Duplicado (...)', and 'GET Obtener Información de la API'. The main panel displays a POST request to 'Crear Envío con ID Duplicado (400)'. The 'Body' tab shows a JSON response with the key 'detail': "Ya existe un envío con ID 1". The status bar at the bottom indicates a 400 Bad Request response.

### Obtener información de la API:



This screenshot shows a successful GET request to 'Obtener Información de la API'. The response body is a JSON object with keys 'mensaje', 'version', and 'documentacion', containing values 'API de Gestión de Envíos', '1.0.0', and '/docs' respectively. The status bar at the bottom indicates a 200 OK response.

## 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 Envio.
- 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 Envio, 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.