Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

T.U.I.A

Redes de Datos

Trabajo Práctico Final

2025

Autora:

Nombre y Apellido	Nº de Legajo
Sol Kidonakis	K-0624/6
Tomas Rodríguez Griñó	R-4643/4

Índice

1. Introducción
2. Descripción del Archivo de Datos (Etapa 1)
   2.1. Elección del Archivo JSON
   2.2. Estructura y Propiedades de los Datos
3. Desarrollo del Servidor API (Etapa 2)
   3.1. Tecnologías Utilizadas
   3.2. Funcionalidades Implementadas
   3.3. Gestión de Datos
4. Desarrollo del Cliente API (Etapa 3)
   4.1. Tecnologías Utilizadas
   4.2. Interfaz y Funcionalidades
5. Configuraciones de Seguridad (Etapa 4)
   5.1. Autenticación Basic
   5.2. Control de Acceso Basado en Roles
   5.3. Limitación de Solicitudes (Rate Limiting)
6. Comunicación entre Hosts (Etapa 5)
   6.1. Topología de Red
   6.2. Configuración de Red de los Hosts
   6.3. Configuración de Firewall en el Servidor Linux
   6.4. Modificación del Cliente API
   6.5. Verificación y Resultados de las Pruebas
7. Conclusión

1. Introducción

El presente informe detalla el desarrollo e implementación del "Proyecto Final: Comunicación API Cliente-Servidor", enmarcado en la materia Redes de Datos. El objetivo principal de este proyecto fue establecer una comunicación fluida y segura entre dos sistemas informáticos distintos, a través de una **API RESTful**.

Una **API (Application Programming Interface)**, o Interfaz de Programación de Aplicaciones, es un conjunto de definiciones y protocolos que permite a diferentes aplicaciones de software comunicarse entre sí. Funciona como un contrato de servicio, definiendo cómo una aplicación puede solicitar funcionalidades o datos a otra, y cómo debe esperar la respuesta. En esencia, una API es un "intermediario" que traduce las peticiones de un sistema a un formato que otro sistema puede entender, ejecutar y responder.

En el contexto de este proyecto, he implementado una API con una arquitectura **Cliente-Servidor**. Esto significa que:

- Un **servidor API** actúa como proveedor de servicios, gestionando un conjunto de datos (en este caso, información sobre los Premios Nobel almacenada en un archivo JSON) y exponiendo diversas funcionalidades (como consultar, crear, actualizar o eliminar datos).
- Un **cliente API** actúa como consumidor, enviando solicitudes al servidor para utilizar esas funcionalidades o acceder a los datos.

La funcionalidad principal de esta API es permitir la interacción programática con el dataset de los Premios Nobel. Esto incluye:

- **Consultas (GET):** Acceder a la información general de la API, buscar premios por año o categoría, obtener la motivación de un premio específico, o buscar laureados por nombre.

- **Modificaciones (POST, PUT, DELETE):** Realizar cambios en la base de datos de premios, como crear nuevos registros, actualizar los existentes o eliminarlos. Estas operaciones, al ser críticas, están protegidas por mecanismos de seguridad.

Este proyecto permite integrar y aplicar conceptos fundamentales de redes, como el formato de datos (JSON) para la representación e intercambio de información, la implementación de APIs RESTful para la interacción entre sistemas, la configuración de mecanismos de autenticación y control de acceso para la seguridad, y la limitación de solicitudes para la protección del servicio. A su vez, se integraron conceptos de redes como la configuración IP, máscaras de subred, gateways y firewalls, esenciales para establecer la comunicación efectiva entre hosts en una red.

2. Descripción del Archivo de Datos (Etapa 1)

2.1. Elección del Archivo JSON

Para este proyecto, se ha utilizado un archivo JSON que contiene información sobre los Premios Nobel. Este archivo, denominado nobel_prizes.json, se obtiene de la API pública de Nobel Prize (https://api.nobelprize.org/v1/prize.json). La elección de este dataset se basó en su estructura anidada,su gran cantidad de datos y la riqueza de estos, lo que permite implementar diversas operaciones de consulta y modificación.

El módulo data_handler.py es el encargado de gestionar este archivo, permitiendo su descarga desde la URL oficial y su carga en memoria para que el servidor API pueda trabajar con él.

2.2. Estructura y Propiedades de los Datos

El archivo nobel_prizes.json contiene un objeto raíz con una propiedad "prizes", que es una lista de objetos. Cada objeto en esta lista representa un Premio Nobel individual.

Cantidad de Objetos: El archivo cargado contiene un total de **676 objetos (premios Nobel)**, lo que proporciona una base de datos considerable para las operaciones de la API.

Estructura de un Objeto 'Premio' (Ejemplo Típico):

Cada premio es un diccionario con las siguientes propiedades principales:

- **year**:
  - ➢ **Tipo de dato:** string
  - ➢ **Descripción:** El año en que se otorgó el premio.
  - ➢ **Ejemplo:** "2024"
- **category**:
  - ➢ **Tipo de dato:** string
  - ➢ **Descripción:** La categoría del premio (ej. "physics", "chemistry", "peace", "medicine", "literature", "economics").
  - ➢ **Ejemplo:** "chemistry"
- **laureates**:
  - ➢ **Tipo de dato:** list de objetos LaureateBase (opcional)
  - ➢ **Descripción:** Una lista de diccionarios, donde cada diccionario representa a un laureado asociado a ese premio. Puede ser null si el premio no tiene laureados asignados (aunque es raro en los datos reales).
- **overallMotivation**:
  - ➢ **Tipo de dato:** string (opcional)
  - ➢ **Descripción:** Una cadena de texto que describe la motivación general del premio. Puede ser null.
  - ➢ **Ejemplo:** "for the discovery of a new form of energy"

Estructura de un Objeto 'Laureado' (Anidado dentro de 'laureates'):

Cada laureado es un diccionario con las siguientes propiedades:

- **id**:
  - ➢ **Tipo de dato:** string (opcional en la entrada, pero generado por el servidor si no se proporciona)

➢ **Descripción:** Un identificador único para el laureado.
➢ **Ejemplo:** "1039" o "solkidonakis2025"

- **firstname**:
  - ➢ **Tipo de dato:** string
  - ➢ **Descripción:** El nombre del laureado.
  - ➢ **Ejemplo:** "Marie"
- **surname**:
  - ➢ **Tipo de dato:** string (opcional)
  - ➢ **Descripción:** El apellido del laureado. Puede ser null.
  - ➢ **Ejemplo:** "Curie"
- **motivation**:
  - ➢ **Tipo de dato:** string (opcional)
  - ➢ **Descripción:** La motivación específica del laureado para ese premio. Puede ser null.
  - ➢ **Ejemplo:** "for her joint research on the radiation phenomena"
- **share**:
  - ➢ **Tipo de dato:** string (opcional)
  - ➢ **Descripción:** La proporción del premio que recibió el laureado (ej. "1" para premio completo, "1/2" para compartido). Puede ser null.
  - ➢ **Ejemplo:** "1"

Decisiones Tomadas en la Etapa 1:

- Se optó por el dataset de los Premios Nobel por su relevancia y estructura.
- Se desarrolló el módulo data_handler.py para encapsular la lógica de descarga, carga y consulta de los datos, promoviendo la modularidad y facilitando la gestión del archivo JSON.
- Se incluyó una función describe_data_structure para generar una descripción programática de la estructura.

3. Desarrollo del Servidor API (Etapa 2)

El servidor API, implementado en server_api.py, es el componente central que expone las funcionalidades de consulta y modificación de los datos de los Premios Nobel.

3.1. Tecnologías Utilizadas

- **FastAPI:** Un framework web moderno y rápido para construir APIs con Python, basado en estándares abiertos como OpenAPI (Swagger) y JSON Schema. Su uso de Pydantic para la validación de datos simplifica el desarrollo y asegura la consistencia.
- **Uvicorn:** Un servidor ASGI (Asynchronous Server Gateway Interface) de alto rendimiento, utilizado para ejecutar la aplicación FastAPI.
- **Pydantic:** Utilizado a través de FastAPI para definir los modelos de datos (models.py), lo que permite la validación automática de los datos de entrada y salida, y la generación de documentación interactiva de la API.

3.2. Funcionalidades Implementadas

El servidor expone una serie de endpoints HTTP para interactuar con los datos:

- **Endpoints de Información y Estado (GET):**
  - ➢ /: Proporciona el estado general de la API (versión, total de premios, seguridad habilitada).
  - ➢ /security/info: Detalla la configuración de seguridad (tipo de autenticación, límites de tasa, endpoints protegidos y roles requeridos).
- **Endpoints de Consulta (GET - Públicos):**
  - ➢ /prizes: Retorna todos los premios Nobel disponibles.
  - ➢ /prizes/year/{year}: Retorna premios de un año específico.
  - ➢ /prizes/category/{category}: Retorna premios de una categoría específica.
  - ➢ /prizes/motivation/{year}/{category}: Retorna la motivación general de un premio.

    - ➢ /laureates/search?firstname={firstname}&surname={surname} : Busca premios por nombre y apellido de laureado.
    - ➢ /laureates/{year}/{category}: Obtiene los laureados de un premio específico.
- **Endpoints de Modificación (POST, PUT, DELETE - Protegidos):**
    - ➢ POST /prizes: Crea un nuevo premio Nobel. Requiere autenticación.
    - ➢ PUT /prizes/{year}/{category}: Actualiza un premio Nobel existente. Requiere autenticación.
    - ➢ DELETE /prizes/{year}/{category}: Elimina un premio Nobel. Requiere autenticación y **permisos de administrador**.

3.3. Gestión de Datos

Un aspecto fundamental del desarrollo del servidor API es la definición de los **modelos de datos**, implementados en el archivo models.py utilizando la librería **Pydantic**.

Los modelos de Pydantic permiten definir la **estructura y el tipo de los datos** que la API espera recibir (como entrada) y los que enviará (como salida).

Cómo se aplican los modelos al caso:

1. **Validación Automática de Entrada:** Cuando un cliente envía una petición POST o PUT a la API, FastAPI utiliza los modelos de Pydantic (PrizeCreate, PrizeUpdate) para **validar automáticamente** los datos JSON recibidos. Si los datos no cumplen con la estructura o los tipos definidos en el modelo (ej., un campo requerido falta, o un tipo de dato es incorrecto), FastAPI generará automáticamente un error 422 Unprocessable Entity antes de que el código de la función se ejecute. Esto garantiza la integridad de los datos que procesa el servidor.
2. **Serialización Automática de Salida:** Cuando el servidor retorna datos (ej., una lista de premios en un GET /prizes), FastAPI utiliza los modelos de Pydantic (PrizeBase, LaureateBase) para **serializar**

automáticamente los objetos Python a formato JSON. Esto asegura que la respuesta enviada al cliente siempre tenga la estructura definida en los modelos, facilitando al cliente la interpretación de los datos.

3. **Documentación Interactiva (OpenAPI/Swagger UI):** Una de las mayores ventajas de usar Pydantic con FastAPI es que los modelos se utilizan para **generar automáticamente la documentación interactiva de la API** (disponible en /docs). Esta documentación muestra claramente qué campos espera cada endpoint, sus tipos de datos, si son opcionales o requeridos, y ejemplos.

Decisiones de Diseño del Servidor:

- La elección de FastAPI permite una rápida implementación y la generación automática de documentación interactiva (/docs).
- La carga de datos en memoria mejora el rendimiento de las consultas.
- La persistencia a un archivo JSON local simplifica el despliegue sin necesidad de una base de datos compleja.
- La gestión de errores con HTTPException proporciona respuestas estandarizadas y claras a los clientes.

4. Desarrollo del Cliente API (Etapa 3)

El cliente API, implementado en client_api.py, es una aplicación de línea de comandos diseñada para interactuar con el servidor API de Premios Nobel.

4.1. Tecnologías Utilizadas

- **requests (Librería de Python):** Se utiliza para realizar todas las solicitudes HTTP (GET, POST, PUT, DELETE) al servidor API. Es una librería estándar y robusta para peticiones web en Python.

4.2. Interfaz y Funcionalidades

El cliente presenta un menú interactivo en la terminal que permite al usuario seleccionar diversas operaciones:

- **Consultas (GET):** Permite obtener información general de la API, información de seguridad, todos los premios, premios por año/categoría, motivaciones, y buscar laureados.
- **Modificaciones (POST, PUT, DELETE):** Permite crear, actualizar y eliminar premios Nobel. Estas operaciones solicitan credenciales de autenticación al usuario antes de enviar la petición al servidor.
- **Manejo de Respuestas:** Incluye funciones auxiliares (handle_response, print_json_response) para interpretar y mostrar las respuestas del servidor de manera legible, incluyendo mensajes de éxito, errores y detalles JSON.
- **Autenticación:** La función get_credentials() solicita el nombre de usuario y la contraseña, y los envía como parte de la autenticación Basic en las solicitudes HTTP protegidas.

Decisiones de Diseño del Cliente:

- Se optó por una interfaz de línea de comandos para cumplir con los requisitos del proyecto y mantener la simplicidad.
- El manejo centralizado de respuestas HTTP (handle_response) mejora la consistencia y la depuración.
- La separación de las funciones de operación (GET, POST, PUT, DELETE) en funciones dedicadas hace el código más modular y fácil de entender.

5. Configuraciones de Seguridad (Etapa 4)

La seguridad es un pilar fundamental del proyecto, implementada en el módulo security_config.py e integrada en el server_api.py. Se abordaron dos aspectos principales: autenticación y limitación de solicitudes.

5.1. Autenticación Basic

- **Implementación:** Se utiliza el esquema HTTPBasic de FastAPI junto con una función de dependencia get_current_user. Esta función verifica las credenciales (username, password) proporcionadas en el encabezado Authorization de la petición HTTP contra un diccionario de usuarios predefinidos en USERS.
- **Credenciales:**
    - user: username="user", password="user123", role="user"
    - admin: username="admin", password="admin123", role="admin"
- **Protección de Endpoints:** Los endpoints POST /prizes, PUT /prizes/{year}/{category} y DELETE /prizes/{year}/{category} requieren autenticación, lo que significa que solo usuarios con credenciales válidas pueden acceder a ellos. Si las credenciales son incorrectas, se retorna un error 401 Unauthorized.

5.2. Control de Acceso Basado en Roles

- **Implementación:** Se definió una función de dependencia require_admin que, además de autenticar al usuario, verifica que su role sea "admin".
- **Endpoints Protegidos por Rol:** El endpoint DELETE /prizes/{year}/{category} está específicamente protegido por require_admin. Esto asegura que solo los usuarios con el rol de "admin" puedan eliminar premios. Si un usuario con rol "user" intenta eliminar un premio, recibirá un error 403 Forbidden.

5.3. Limitación de Solicitudes (Rate Limiting)

- **Implementación:** Se utilizó la librería slowapi, que integra un Limiter basado en la dirección IP remota del cliente.
- **Límites Definidos:** Se establecieron diferentes límites de tasa en el diccionario RATE_LIMITS:
    - "default": "100/minute" (para endpoints públicos)
    - "user": "50/minute" (para usuarios autenticados con rol "user")
    - "admin": "200/minute" (para usuarios autenticados con rol "admin")

    - "strict": "10/minute" (para operaciones sensibles como POST/PUT/DELETE)
- **Configuración en FastAPI:** El Limiter se adjunta a la aplicación FastAPI y se configura un manejador de excepciones para RateLimitExceeded, retornando un error si se supera el límite.
- **Aplicación:** Cada endpoint del servidor tiene un decorador @limiter.limit() que aplica el límite de tasa correspondiente.

Decisiones de Seguridad:

- La autenticación Basic es un método sencillo y efectivo para este tipo de proyecto.
- La implementación de roles (user, admin) permite una granularidad en el control de acceso, siendo el rol de administrador el único con permisos para eliminar datos.
- El rate limiting es crucial para proteger la API contra ataques de fuerza bruta o de denegación de servicio básicos, asegurando la disponibilidad del servicio.

6. Comunicación entre Hosts (Etapa 5)

Esta etapa crucial valida la capacidad de la aplicación para funcionar en un entorno de red real, con el servidor y el cliente ejecutándose en máquinas separadas.

6.1. Topología de Red

La configuración de red consistió en dos notebooks conectadas a la misma red local a través de Wi-Fi, compartiendo el mismo router/punto de acceso.

- **Notebook Servidor:** Una máquina con sistema operativo **Linux**.
- **Notebook Cliente:** Una máquina con sistema operativo **Windows** (la notebook de testeo).
- Ambas máquinas se encuentran en la misma subred, lo que permite la comunicación directa entre ellas.

6.2. Configuración de Red de los Hosts

Para que dos dispositivos puedan comunicarse en una red, cada uno necesita una **Dirección IP (Internet Protocol)** única dentro de esa red. La Dirección IP es como la dirección postal de un dispositivo en la red. En este caso, estamos utilizando direcciones IPv4, que son números de 32 bits representados en cuatro octetos (ej., 192.168.100.12).

La **Máscara de Subred** (ej., 255.255.255.0 o /24) es un valor que le indica a un dispositivo qué parte de una dirección IP corresponde a la red y qué parte corresponde al host. Si dos dispositivos están en la "misma subred", significa que la parte de su dirección IP que identifica la red es idéntica, lo que les permite comunicarse directamente sin necesidad de un router para enrutar el tráfico entre ellos. En nuestro caso, ambas máquinas tienen IPs que comienzan con 192.168.100., y una máscara de subred 255.255.255.0, lo que las coloca en la misma subred 192.168.100.0/24.

El **Gateway (Puerta de Enlace Predeterminada)** es la dirección IP del dispositivo (generalmente un router) que permite a los equipos de la red local comunicarse con redes externas (como Internet) o con otras subredes. Si un dispositivo necesita enviar datos a una IP que no está en su misma subred, los envía al gateway para que este los reenvíe. En nuestra configuración, el gateway es 192.168.100.1, que es la dirección del router Wi-Fi al que ambas notebooks están conectadas.

Se registraron las siguientes configuraciones de red para cada equipo:

- **Notebook Servidor (Linux - rtadmin-ThinkPad-E16-Gen-2):**
  - ➢ **Interfaz Principal:** wlp0s20f3 (Wi-Fi)
  - ➢ **Dirección IPv4:** 192.168.100.12
  - ➢ **Máscara de Subred:** 255.255.255.0 (/24)
  - ➢ **Puerta de Enlace Predeterminada (Gateway):** 192.168.100.1

Evidencia (ip addr y ip route):

```
rtadmin@rtadmin-ThinkPad-E16-Gen-2:~$ ip addr
...
3: wlp0s20f3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 44:a3:bb:9c:e7:08 brd ff:ff:ff:ff:ff:ff
    altname wlx44a3bb9ce708
    inet 192.168.100.12/24 brd 192.168.100.255 scope global dynamic noprefixroute wlp0s20f3
...
rtadmin@rtadmin-ThinkPad-E16-Gen-2:~$ ip route
default via 192.168.100.1 dev wlp0s20f3 proto dhcp src 192.168.100.12 metric 600
192.168.100.0/24 dev wlp0s20f3 proto kernel scope link src 192.168.100.12 metric 600
```

- **Notebook Cliente (Testeo - Windows):**
  - ➢ **Interfaz Principal:** Wi-Fi
  - ➢ **Dirección IPv4:** 192.168.100.5
  - ➢ **Máscara de Subred:** 255.255.255.0
  - ➢ **Puerta de Enlace Predeterminada (Gateway):** 192.168.100.1

Evidencia (ipconfig):

```
C:\Users\solki>ipconfig
...
Adaptador de LAN inalámbrica Wi-Fi:
    Sufijo DNS específico para la conexión. . :
    Dirección IPv4. . . . . . . . . . . . . . : 192.168.100.5
    Máscara de subred . . . . . . . . . . . . : 255.255.255.0
    Puerta de enlace predeterminada . . . . . : 192.168.100.1
```

6.3. Configuración de Firewall en el Servidor Linux

Un **Firewall** es un sistema de seguridad de red que controla el tráfico de red entrante y saliente basándose en un conjunto de reglas de seguridad

preestablecidas. Actúa como una barrera entre una red de confianza y una que no lo es, protegiendo los sistemas de accesos no autorizados. En este proyecto, el firewall del servidor Linux podría haber bloqueado las conexiones entrantes del cliente si no se configuraba adecuadamente.

Para permitir que el cliente se conectara al servidor, fue necesario configurar el firewall en la notebook Linux (servidor) para abrir el puerto 8001 para el tráfico TCP entrante. Se utilizó ufw (Uncomplicated Firewall), una interfaz sencilla para gestionar netfilter en Linux.

Comandos Ejecutados:

```
sudo ufw allow 8001/tcp
sudo ufw enable
```

Estado del Firewall (sudo ufw status):

```
rtadmin@rtadmin-ThinkPad-E16-Gen-2:~$ sudo ufw status
Estado: activo

Hasta                      Acción      Desde
-----                      ------      -----
8001/tcp                   ALLOW       Anywhere
8001/tcp (v6)              ALLOW       Anywhere (v6)
```

Justificación: Esta regla es indispensable para que las peticiones del cliente (desde 192.168.100.5) puedan alcanzar el servidor API que escucha en el puerto 8001 en 192.168.100.12.

6.4. Modificación del Cliente API

Para que el cliente API pudiera comunicarse con el servidor remoto, se modificó la variable BASE_URL en el archivo client_api.py para apuntar a la dirección IP del servidor Linux:

Línea Modificada en client_api.py:
BASE_URL = "http://192.168.100.12:8001"

- **Justificación:** Sin este cambio, el cliente intentaría conectarse a un servidor en su propia máquina (localhost), lo cual no es el objetivo de la comunicación entre hosts.

6.5. Verificación y Resultados de las Pruebas

Se realizaron pruebas exhaustivas desde la notebook cliente para verificar la conectividad y la funcionalidad de la API.

- **Prueba de Conectividad (ping):**

Desde la notebook cliente (Windows), se ejecutó un ping a la IP del servidor Linux:

```
PS C:\Users\solki> ping 192.168.100.12

Haciendo ping a 192.168.100.12 con 32 bytes de datos:
Respuesta desde 192.168.100.12: bytes=32 tiempo=74ms TTL=64
Respuesta desde 192.168.100.12: bytes=32 tiempo=10ms TTL=64
Respuesta desde 192.168.100.12: bytes=32 tiempo=24ms TTL=64
Respuesta desde 192.168.100.12: bytes=32 tiempo=28ms TTL=64

Estadísticas de ping para 192.168.100.12:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
```

Resultado: El ping fue exitoso, confirmando la conectividad básica de red entre ambas máquinas.

- **Pruebas de Funcionalidad de la API desde el Cliente:**
  ➢ **Consulta GET (Información de la API - Opción 0):**

- **Resultado:** Éxito (Código de estado: 200). La respuesta JSON ("status": "active", "total_prizes": 679, "security_enabled": true) confirmó que el cliente se conectó exitosamente al servidor remoto y obtuvo la información esperada.
- **Log del servidor:** INFO: 192.168.100.5:59580 - "GET / HTTP/1.1" 200 OK

➢ **Consulta GET (Premios por Año - Opción 2, año 2024):**

- **Resultado:** Éxito (Código de estado: 200). Se recibieron los premios correspondientes al año 2024, incluyendo los premios de prueba creados previamente.
- **Log del servidor:** INFO: 192.168.100.5:59644 - "GET /prizes/year/2024 HTTP/1.1" 200 OK

➢ **Operación POST (Crear Premio - Opción 7):**

- **Credenciales:** admin / admin123
- **Resultado:** Éxito (Código de estado: 201 Created). Se creó un nuevo premio para 2025/medicine con laureado "Sol Kidonakis".
- **Log del servidor:** INFO: 192.168.100.5:59675 - "POST /prizes HTTP/1.1" 201 Created
- **Verificación:** Una posterior consulta GET al año 2025 confirmó la presencia del nuevo premio.

➢ **Operación PUT (Actualizar Premio - Opción 8):**

- **Credenciales:** admin / admin123
- **Resultado:** Éxito (Código de estado: 200 OK). El premio de 2025/medicine fue actualizado a 2026/medicine.
- **Log del servidor:** INFO: 192.168.100.5:59690 - "PUT /prizes/2025/medicine HTTP/1.1" 200 OK
- **Verificación:** Una posterior consulta GET al año 2026 confirmó la actualización.

➢ **Operación DELETE (Eliminar Premio - Opción 9 - Prueba de Seguridad):**

- **Intento 1 (con credenciales user / user123):**

        - **Resultado:** ERROR (Código de estado: 403 Forbidden). El servidor respondió: "detail": "Se requieren permisos de administrador".
        - **Log del servidor:** INFO: 192.168.100.5:59714 - "DELETE /prizes/2026/medicine HTTP/1.1" 403 Forbidden
        - **Conclusión de Seguridad:** Esta es una prueba crucial y exitosa de que el control de acceso basado en roles funciona correctamente. El servidor recibió la solicitud, autenticó al usuario, pero denegó la operación debido a la falta de permisos de administrador para la acción de eliminación.
    - **Intento 2 (con credenciales admin / admin123):** Se realizó y fue exitosa la eliminación.
        - **Conclusión de Seguridad:** Se demostró que el rol de administrador tiene los permisos correctos para eliminar.

7. Conclusión

Este proyecto logro exitosamente el desarrollo e implementación de una comunicación API Cliente-Servidor robusta y segura. Se han cumplido los requisitos del trabajo:

- **Etapa 1:** Se seleccionó y se describió detalladamente la estructura de los datos de los Premios Nobel.
- **Etapa 2:** Se implementó un servidor API completo utilizando FastAPI, capaz de gestionar consultas y modificaciones (CRUD) sobre los datos.
- **Etapa 3:** Se desarrolló un cliente API interactivo en Python que puede consumir todas las funcionalidades del servidor.
- **Etapa 4:** Se integraron configuraciones de seguridad esenciales, incluyendo autenticación Basic con roles de usuario (user, admin), control de acceso basado en roles (protegiendo la eliminación a solo

administradores), y limitación de solicitudes (rate limiting) para proteger la API.

- **Etapa 5:** Se estableció y verificó la comunicación entre dos hosts distintos (una máquina Linux como servidor y una Windows como cliente), demostrando la funcionalidad de la API en un entorno de red real y validando todas las configuraciones de red y firewall.

Las pruebas realizadas confirman la solidez de la implementación y la correcta aplicación de las políticas de seguridad.