

TECNICATURA UNIVERSITARIA EN CIBERSEGURIDAD

DESARROLLO DE SOFTWARE SEGURO

Trabajo Practico Final

**Diseño e Implementación de APIs Seguras con Flask,
FastAPI y OpenAPI - Primer Entrega**

Alumnos

Cabrera, Ricardo Martin

Deparsia, Ignacio

Quaglia, Tomas

Profesor

Licenciado Juan Pablo Villalba

Villa Real (CABA), Merlo (San Luis), Mendoza Capital (Mendoza)

Año 2025

Índice

Resumen	3
Introducción	3
1.Desarrollo – Análisis Teórico	
1.1 Métodos de autenticación en APIs	4
1.1.1 Autenticación Básica	4
1.1.2 Autenticación por IP	4
1.1.3 Autenticación por API Key	4
1.1.4 OAuth 2.0 y JWT	4
1.2 Comparación entre Flask y FastAPI	5
1.3 Importancia de OpenAPI	5
2. Implementación Práctica	5
Conclusión	10
Referencias Bibliográficas	11
Bibliografía	11

Resumen

Este informe corresponde a la primera de tres entregas del Trabajo Práctico Final de la materia. En esta etapa nos vamos a enfocar en los fundamentos de la autenticación y en la configuración inicial de APIs, sentando las bases para construir servicios web seguros y bien documentados.

Vamos a analizar y comparar distintos métodos de autenticación que suelen usarse en APIs. Empezaremos por los enfoques más simples y tradicionales, como la autenticación básica, por dirección IP y mediante API Keys, evaluando sus ventajas, limitaciones y casos de uso. También vamos a introducir mecanismos más modernos como OAuth 2.0 y JSON Web Tokens (JWT), que se usan mucho hoy en día por su compatibilidad con arquitecturas sin estado.

Desde el lado práctico, el trabajo incluye la creación de APIs utilizando dos frameworks muy populares de Python: Flask y FastAPI. En ambos casos vamos a implementar autenticación básica en rutas protegidas y a documentar las APIs usando OpenAPI con Swagger UI. Esto nos va a permitir ver en acción cómo se integran los distintos métodos de autenticación y cómo se generan interfaces interactivas para probar las rutas.

Además, vamos a evaluar las diferencias entre Flask y FastAPI, prestando atención a su rendimiento y a cómo se adaptan para integrar autenticación y documentación de forma sencilla.

En resumen, esta primera entrega busca combinar la teoría y la práctica para desarrollar un enfoque integral en el diseño de APIs seguras y bien documentadas. Todo el código fuente está disponible en un repositorio público de GitHub, como evidencia práctica de lo trabajado.

Introducción

Esta primera entrega tiene como propósito analizar los distintos enfoques de autenticación utilizados en APIs, desde los métodos clásicos como la autenticación básica, a técnicas más avanzadas como OAuth 2.0 y JWT (usadas en aplicaciones más modernas). También evaluaremos los aspectos arquitectónicos y de rendimiento de dos frameworks populares de Python: Flask y FastAPI, con un enfoque especial en sus capacidades para integrar mecanismos de autenticación y documentación de APIs mediante OpenAPI.

En el componente práctico complementaremos esta teoría mediante proyectos "funcionales", implementando rutas protegidas y generando documentación interactiva con Swagger UI. Entonces, el objetivo de la teoría y práctica de este trabajo es permitir establecer un enfoque integral en el diseño de APIs seguras y documentadas, para lograr una buena base en los siguientes trabajos.

Desarrollo

1. Análisis Teórico

1.1 Métodos de Autenticación en APIs

1.1.1 Autenticación Básica

La autenticación básica es la forma más simple de autenticación, ya que el cliente envía un nombre de usuario y una contraseña codificados en Base64 dentro del encabezado HTTP Authorization. Su principal ventaja es que es bastante cómoda y fácil de implementar y es compatible con múltiples lenguajes y frameworks. Sin embargo, su gran limitación es la seguridad: ya que no se puede utilizar sin cifrado HTTPS, porque las credenciales pueden ser interceptadas fácilmente por terceros. Por este motivo, su uso debe reservarse para entornos controlados o complementarse con una capa de cifrado segura.

1.1.2 Autenticación por IP

Esta forma de autenticación permite restringir el acceso a una API en función de la dirección IP del cliente. Es muy utilizada en sistemas internos o redes corporativas donde solo ciertos dispositivos pueden conectarse. Su mayor fortaleza es el control preciso del acceso en entornos estáticos (puede ser interesante usarlo dentro de una empresa, donde todos necesitan estar conectados al mismo programa); sin embargo, presenta importantes limitaciones, como la vulnerabilidad ante técnicas de suplantación de IP (IP spoofing) y su ineficiencia en contextos donde las IPs de los clientes cambian frecuentemente, como ocurre con conexiones móviles o servicios en la nube (puede traer problemas cuando se utiliza fuera del “entorno controlado”).

1.1.3 Autenticación por API Key

La autenticación por API Key consiste en proporcionar al cliente un identificador único que debe incluir en cada solicitud. Este método permite un control más minucioso del acceso y facilita la identificación de cada cliente que consume la API. No obstante, al igual que con la autenticación básica, las claves deben protegerse adecuadamente y rotarse periódicamente, ya que, si se filtran pueden ser usadas por terceros no autorizados. Se podría utilizar un método en el cual genere una Key distinta cada cierto tiempo, y esa “contraseña” ingresarla en la solicitud (como Google Authenticator).

1.1.4 OAuth 2.0 y JWT

OAuth 2.0 es un protocolo de autorización que permite a las aplicaciones acceder a recursos protegidos en nombre del usuario sin necesidad de compartir las credenciales. Funciona mediante flujos específicos (como el Authorization Code Flow), y permite la delegación segura de permisos entre aplicaciones. Por ejemplo, un usuario puede autorizar a una app de terceros a acceder a su cuenta de Google sin compartir su contraseña.

JWT (JSON Web Token) es un estándar que define un formato compacto y seguro para transmitir información entre partes como un objeto JSON firmado digitalmente. Los JWT se usan comúnmente en autenticación sin estado (stateless authentication), ya que una vez emitido, el token puede ser verificado por el servidor sin necesidad de mantener sesiones activas. El token puede incluir datos útiles (como el ID de usuario o sus permisos), y es validado por su firma, evitando manipulaciones.

1.2 Comparación entre Flask y FastAPI

Flask es un microframework escrito en Python, simple y flexible. Permite construir aplicaciones web con rapidez y control total sobre su arquitectura. No incluye muchas herramientas por defecto, lo que lo hace ideal para proyectos pequeños o cuando se requiere una implementación personalizada desde cero. Funciona de manera sincrónica.

FastAPI, por otro lado, es un framework moderno que también utiliza Python, pero está diseñado específicamente para construir APIs de forma eficiente, utilizando programación asíncrona y tipos de datos estrictos gracias a Pydantic. Además, ofrece soporte automático para documentación con OpenAPI y Swagger UI. Está construido sobre Starlette y tiene mejor rendimiento que Flask en escenarios grandes y de mucho flujo.

1.3 Importancia de OpenAPI

OpenAPI es una especificación estándar para describir APIs RESTful. Permite definir las rutas, métodos, parámetros, respuestas y autenticación de una API de forma estructurada, normalmente en archivos .yaml o .json. Gracias a esta estandarización, es posible generar automáticamente documentación interactiva mediante herramientas como Swagger UI o Redoc, facilitar pruebas, integrar clientes o SDKs, y mejorar la colaboración entre desarrolladores y equipos. En el contexto de desarrollo seguro, OpenAPI ayuda a validar que las APIs expuestas cumplen con los requisitos definidos y evita exposiciones innecesarias.

2. Implementación Práctica

Para la realización de las tareas consignadas, primero hemos realizado la verificación de los elementos operativos necesarios en nuestro entorno de Python. Por lo cual realizamos la instalación de los componentes de FastAPI desde nuestra línea de comandos: `pip install "fastapi[standard]"`

FastAPI es un framework de Python de alto rendimiento para construir APIs REST, especialmente diseñado para la creación de aplicaciones web y microservicios. Utiliza las sugerencias de tipos estándar de Python, la validación de datos, la documentación automática de la API y otros mecanismos para acelerar el desarrollo de aplicaciones.

En resumen, FastAPI es una herramienta que ayuda a los desarrolladores de Python a crear APIs web de forma rápida y eficiente, con la ayuda de la validación automática de

datos, la generación de documentación y otras características avanzadas. Y luego de igual manera realizamos la instalación de los componentes de Flask: `pip install flask`

Flask es un microframework de desarrollo web para Python. Es un marco de trabajo ligero, minimalista y flexible que facilita la creación de aplicaciones web de forma rápida y sencilla. A diferencia de otros frameworks más grandes, Flask ofrece una gran libertad al desarrollador, permitiéndole agregar solo las funcionalidades necesarias y evitar dependencias innecesarias.

También nos informamos que para la ejecución local en entornos Windows, era necesario tener instalado Uvicorn, el cual instalamos con: `pip install uvicorn`

Uvicorn es un servidor ASGI ultrarrápido para Python, ideal para ejecutar aplicaciones FastAPI. Este componente clave, gestiona la comunicación entre la aplicación web y los clientes que envían solicitudes HTTP, permitiendo que la aplicación procese y responda a esas solicitudes de manera eficiente.

El término de Servidor ASGI (Asynchronous Server Gateway Interface) refiere a que es una interfaz que define como un servidor web y un framework web deben interactuar. Uvicorn implementa esta interfaz para permitir la ejecución asíncrona de aplicaciones web, lo que mejora su rendimiento.

Revisamos los componentes con pip show para cada uno de los elementos:

`pip show fastapi`

`pip showuvicorn`

`pip show flask`

```
Microsoft Windows [Versión 10.0.26100.4061]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Rimaca>pip show fastapi
Name: fastapi
Version: 0.115.12
Summary: FastAPI framework, high performance, easy to learn, fast to code, ready for production
Home-page: https://github.com/fastapi/fastapi
Author:
Author-email: =?utf-8?q?Sebasti=C3=A1n_Ram=C3=ADrez?= <tiangolo@gmail.com>
License:
Location: C:\Users\Rimaca\AppData\Local\Programs\Python\Python313\Lib\site-packages
Requires: pydantic, starlette, typing-extensions
Required-by:

C:\Users\Rimaca>pip show flask
Name: Flask
Version: 3.1.1
Summary: A simple framework for building complex web applications.
Home-page:
Author:
Author-email:
License-Expression: BSD-3-Clause
Location: C:\Users\Rimaca\AppData\Local\Programs\Python\Python313\Lib\site-packages
Requires: blinker, click, itsdangerous, jinja2, markupsafe, werkzeug
Required-by:

C:\Users\Rimaca>pip show uvicorn
Name: uvicorn
Version: 0.34.2
Summary: The lightning-fast ASGI server.
Home-page: https://www.uvicorn.org/
Author:
Author-email: Tom Christie <tom@tomchristie.com>, Marcelo Trylesinski <marcelotryle@gmail.com>
License-Expression: BSD-3-Clause
Location: C:\Users\Rimaca\AppData\Local\Programs\Python\Python313\Lib\site-packages
Requires: click, h11
Required-by: fastapi-cl
```

Imagen de elaboración propia

Luego de estos pasos, ya teniendo todos los componentes asegurados para trabajar, procedimos a revisar los códigos propuestos dados en el trabajo de ejemplo.

Primero usamos el código de FastApi:

```
from fastapi import FastAPI, HTTPException, Depends
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()
security = HTTPBasic()

def verificar_credenciales(credenciales: HTTPBasicCredentials = Depends(security)):
    usuario_correcto = credenciales.username == "admin"
    contrasena_correcta = credenciales.password == "secret"
    if not (usuario_correcto and contrasena_correcta):
        raise HTTPException(status_code=401, detail="Credenciales invalidas")
    return credenciales.username

@app.get("/api/protegida")
def ruta_protegida(usuario: str = Depends(verificar_credenciales)):
    return {"mensaje": f"Acceso concedido a {usuario}!"}
```

Con este código se generó el archivo *tp001.py*, el cual ejecutamos con la siguiente sentencia que ubicamos en una consulta con Google, la cual iniciara el servidor FastApi:

```
uvicorn tp001:app --reload
```

Donde *tp001* es el nombre del archivo Python (tp001.py), y app es la instancia de FastAPI. Esto iniciará un servidor local, típicamente en <http://127.0.0.1:8000>.

```
C:\Users\Rimaca\Desktop>uvicorn tp001:app --reload
INFO:     Will watch for changes in these directories: ['C:\\Users\\Rimaca\\Desktop']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [1600] using WatchFiles
INFO:     Started server process [10428]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Imagen de elaboración propia

Para acceder al endpoint se utiliza /api/protegida con lo cual en un navegador debemos ingresar: <http://127.0.0.1:8000/api/protegida>

El resultado será:



Imagen de elaboración propia

Si ingresamos un usuario y contraseña erróneos, por ejemplo, Nombre de usuario: *NoLoSe* y Contraseña: *NoLoSe*:

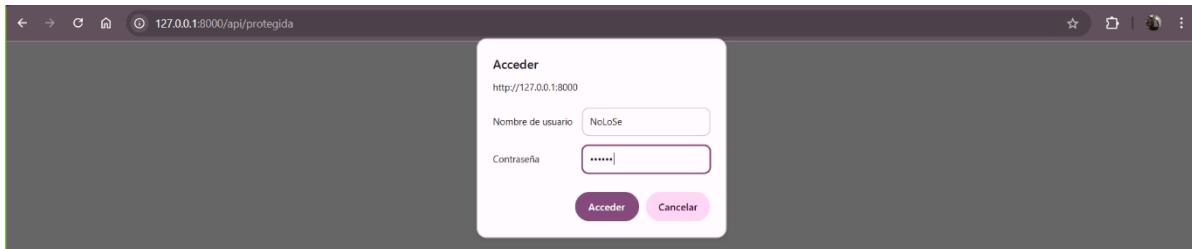


Imagen de elaboración propia



En cambio, si ingresamos como Nombre de usuario: admin y contraseña: secret:



Imagen de elaboración propia

El resultado será:



Imagen de elaboración propia

Documentación Técnica Interactiva

Swagger UI

Para acceder a este informe que se genera de manera automática debemos acceder al link: <http://localhost:8000/docs>. El presente informe también se incluye en el repositorio: [FastAPI - Swagger UI-TP001.pdf](#)

[Authorize](#)
default
[GET /api/protegida Ruta Protegida](#)


Parameters

[Cancel](#)

No parameters

[Execute](#)
[Clear](#)

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/api/protegida' \
  -H 'accept: application/json' \
  -H 'Authorization: Basic YmRtaW46c2VjcmV0'
```

Request URL

<http://127.0.0.1:8000/api/protegida>

Server response

Code Details

200

Response body

```
{
  "mensaje": "Acceso concedido a admin!"
}
```

[Copy](#) [Download](#)

Response headers

```
content-length: 39
content-type: application/json
date: Sun, 25 May 2025 15:33:18 GMT
server: uvicorn
```

Responses

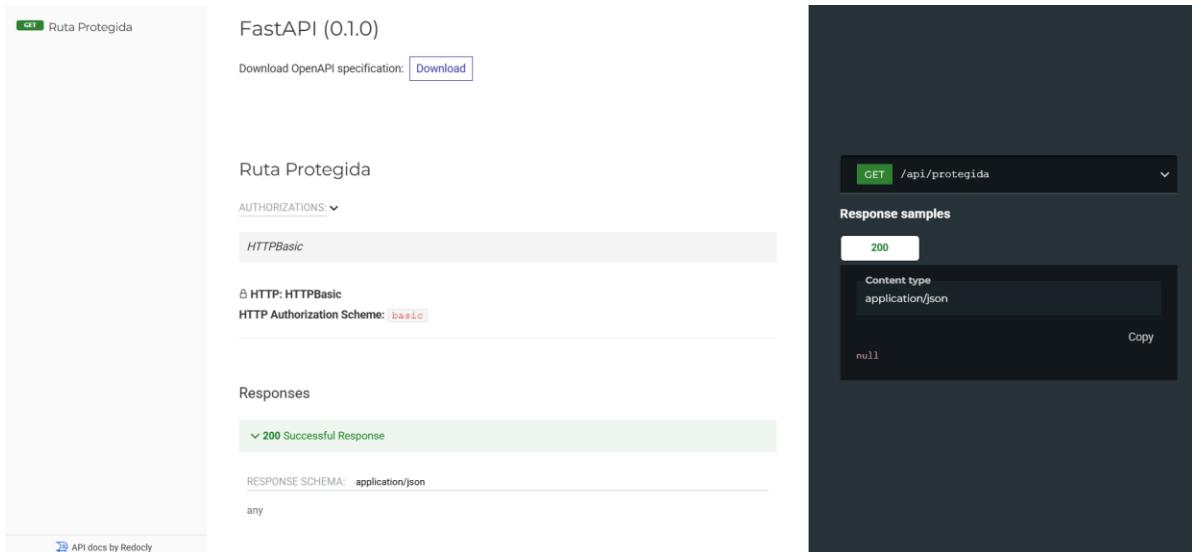
Code Description

[Links](#)
Imagen de elaboración propia

Swagger es una interfaz interactiva para probar endpoints, ver parámetros, respuestas y hacer pruebas. La interactividad es alta ya que puede realizar request, de diseño simple ideal para pruebas y desarrollo.

ReDOC

Para acceder a este informe que se genera de manera automática debemos acceder al link: <http://127.0.0.1:8000/redoc>. El presente informe también se incluye en el repositorio: [FastAPI - ReDoc-TP001.pdf](#)



The screenshot shows a FastAPI (0.1.0) API documentation page. At the top, there's a green bar with the text "Ruta Protegida". Below it, the title "FastAPI (0.1.0)" and a link to "Download OpenAPI specification". A "Download" button is also present. The main content area is titled "Ruta Protegida" and includes sections for "AUTHORIZATIONS:" (HTTPBasic), "Responses" (200 Successful Response), and "RESPONSE SCHEMA: application/json". To the right, a dark-themed panel displays a "Response samples" section for a GET request to "/api/protegida". It shows a 200 status code, a "Content type: application/json" header, and a "null" response body.

Imagen de elaboración propia

ReDOC genera una documentación más detallada y estructurada, excelente para revisión técnica o entrega. Aunque posee una interactividad baja ya que es de solo lectura, su diseño es más profesional y estructurado, haciéndolo ideal para presentaciones y documentación.

Realizamos como segunda parte, la implementación utilizando en esta oportunidad los comandos con Flask. El Código Fuente propuesto es:

```
from flask import Flask, request, jsonify
from functools import wraps

app = Flask(__name__)

def verificar_autenticacion(f):
    @wraps(f)
    def decorador(*args, **kwargs):
        auth = request.authorization
        if not auth or not (auth.username == 'admin' and auth.password == 'secret'):
            return jsonify({"mensaje": "Autenticaci&n fallida!"}), 401
        return f(*args, **kwargs)
    return decorador

@app.route('/api/protegida', methods=['GET'])
@verificar_autenticacion
def api_protegida():
    return jsonify({"mensaje": "Acceso concedido!"})

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

Con este código generamos el archivo `tp002.py`, y en este caso la ejecución se realiza directamente vía Python con: `python tp002.py`

Donde python es nuestro lenguaje de programación y ejecutor de la instancia y `tp002.py` es nuestro archivo a ejecutar. Esto nos iniciara un servidor local típicamente en `http://127.0.0.1:5000`

```
C:\Users\Rimaca\Desktop>python tp002.py
 * Serving Flask app 'tp002'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Imagen de elaboración propia

Para acceder al endpoint se utiliza `/api/protégida` con lo cual en un navegador debemos ingresar: <http://127.0.0.1:5000/api/protégida>

El resultado será:



Imagen de elaboración propia

Si ingresamos un usuario y contraseña erróneos, por ejemplo, Nombre de usuario: NoLoSe y Contraseña: NoLoSe:



Imagen de elaboración propia

El resultado será:

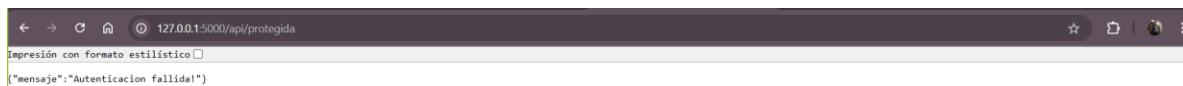


Imagen de elaboración propia

En cambio, si ingresamos como Nombre de usuario: admin y contraseña: secret:



Imagen de elaboración propia

El resultado será:



Imagen de elaboración propia

Documentación Técnica Interactiva

Swagger UI

Para con Flask no hay generación documental integrada, por lo cual debemos usar algún de las tantas interfaces que están disponibles. De entre ellas, elegimos Flask-RESTX para tener API Flask con Swagger UI integrada. Procedemos a instalar la aplicación con:

```
pip install flask-restx
```

Para que podamos obtener el informe, debemos incrustar unas líneas adicionales que habilitan la generación del mismo. Aquí valiéndonos de la ayuda de CHAT-GPT, hemos obtenido el siguiente código:

```
from flask import Flask, request, jsonify, make_response
from functools import wraps
from flask_restx import Api, Resource

app = Flask(__name__)

# 🔒 Configuración de Swagger con autenticación básica
authorizations = {
    'basicAuth': {
        'type': 'basic'
    }
}

api = Api(
    app,
    title="API Segura",
    description="Documentación con Swagger",
    authorizations=authorizations,
    security='basicAuth',
    doc="/docs" # Documentación Swagger disponible en /docs
)

# 📁 Namespace: agrupa los endpoints bajo /protegida/

```

```

ns = api.namespace("protegida", description="Operaciones protegidas")

# 🔒 Decorador de autenticación básica
def verificar_autenticacion(f):
    @wraps(f)
    def decorador(*args, **kwargs):
        auth = request.authorization
        if not auth or not (auth.username == 'admin' and auth.password == 'secret'):
            response = make_response(jsonify({"mensaje": "Autenticación fallida!"}), 401)
            response.headers['WWW-Authenticate'] = 'Basic realm="Login Required"'
            return response
        return f(*args, **kwargs)
    return decorador

# 🔒 Endpoint protegido documentado
@ns.route("/")
class RutaProtegida(Resource):
    @api.doc(security='basicAuth', responses={200: 'OK', 401: 'No autorizado'})
    @verificar_autenticacion
    def get(self):
        return {"mensaje": "Acceso concedido!"}

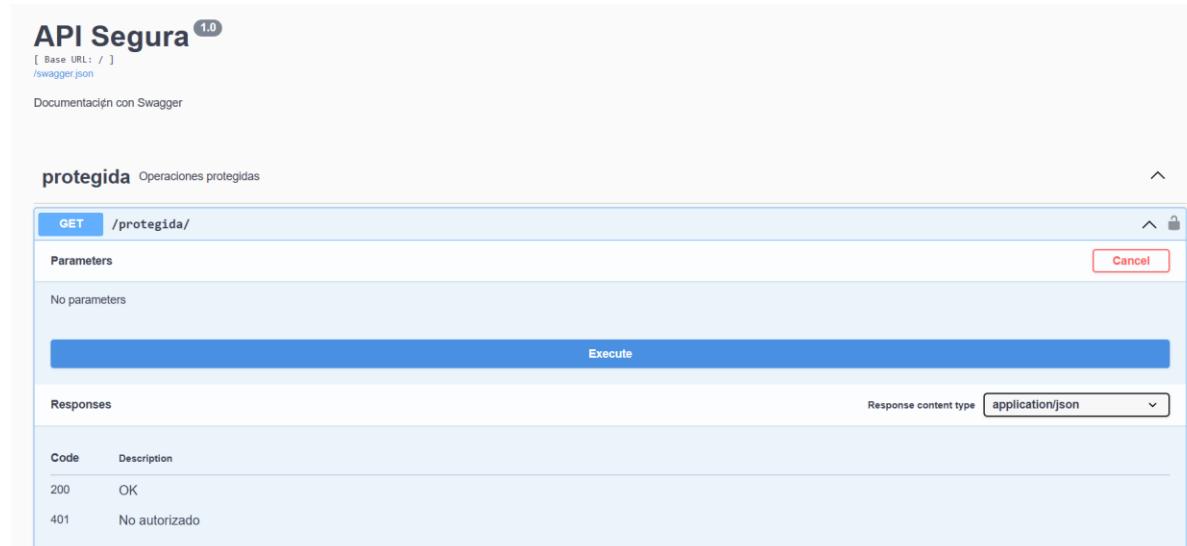
# 🚀 Inicio de la app
if __name__ == '__main__':
    app.run(debug=True)

```

Este código se encuentra disponible en el repositorio: [tp002Documentado.py](#)

Ya con la instalación del complemento y la modificación del código, volvemos a ejecutar el servidor como se hizo con anterioridad. Ahora, podemos acceder a este informe que se genera de manera mediante el siguiente link: <http://localhost:5000/docs>

El presente informe también se incluye en el repositorio: [Flask - Swagger UI-TP02.pdf](#)



The screenshot shows the API Segura 1.0 interface. At the top, it says "API Segura 1.0" and "[Base URL: /] /swagger.json". Below that is a "Documentación con Swagger" section. The main part of the interface is titled "protegida Operaciones protegidas". It shows a "GET /protegida/" operation. Under "Parameters", it says "No parameters". There is a large blue "Execute" button. Under "Responses", there is a table:

Code	Description
200	OK
401	No autorizado

At the bottom right, it says "Response content type application/json".

Imagen de elaboración propia

Swagger es una interfaz interactiva para probar endpoints, ver parámetros, respuestas y hacer pruebas. La interactividad es alta ya que puede realizar request, de diseño simple ideal para pruebas y desarrollo.

A futuro se deberá de tener en cuenta que estas implementaciones son solo con carácter de laboratorio, ya que poseen varias brechas de seguridad, como ser:

- Credenciales de ingreso Harcodeadas, muy fáciles de comprometer porque se ven directamente en el código fuente. Una futura solución sería usar variables de entorno.
- La autenticación básica vía HTTP, que es débil sin control de sesión ni expiración, la cual debería reemplazarse por OAuth2 o JWT.
- Sin transmisión en HTTPS, las credenciales siempre viajan en texto plano, susceptibles a ataques del tipo MITM (Man In The Middle), por lo cual fuera de las experiencias en laboratorio debería forzarse la comunicación con HTTPS.
- Sin protección contra ataques de fuerza bruta ya que se permiten los intentos ilimitados de acceso, con lo cual se debería de agregar un rate limiting.

Es decir que aquí vemos la falta de la utilización de varias Buenas Prácticas que hemos visto a lo largo del cursado.

Conclusión

Después de haber explorado y comparado distintos métodos de autenticación y haberlos implementado en dos frameworks populares como Flask y FastAPI, podemos decir que este primer tramo del trabajo final nos ayudó bastante a entender cómo arrancar una API desde cero, pero con la mirada puesta en la seguridad desde el inicio. No se trata solo de que una API funcione, sino de que esté pensada para proteger los datos que maneja y los usuarios que la consumen.

Durante la parte práctica, meternos en el código y hacer funcionar rutas protegidas nos sirvió mucho para bajar a tierra los conceptos teóricos. Ver cómo se implementa una autenticación básica y cómo se puede documentar con herramientas como Swagger UI nos dio una idea más concreta de cómo es el flujo real de trabajo cuando desarrollamos una API. A la vez, también nos dimos cuenta de las limitaciones que tienen algunas de estas soluciones si no se complementan con buenas prácticas, como el uso de HTTPS, autenticación más robusta (OAuth2, JWT) y protección contra ataques como fuerza bruta.

Como para cerrar, podríamos decir que esta entrega nos dejó una base sólida, tanto teórica como técnica, para seguir avanzando en el desarrollo de APIs seguras. Vimos que la

seguridad es más que solo cuestión de herramientas o librerías, sino que las decisiones que se toman desde el diseño de la aplicación son de suma importancia.

Y lo más importante: pudimos experimentar con todo esto de forma práctica, lo que nos permite prepararnos mejor para las próximas partes del trabajo final.

Referencias Bibliográficas

- Gemini (Modelo de lenguaje de IA). (2025). Interacción con el usuario sobre el análisis y mejora de código Python para una aplicación IVR. [Fecha de la interacción: 9 de abril de 2025]. <https://gemini.google.com/app/983078de7faa96e7?hl=es&pli=1>
- Josefsson, S. (2006). The Base16, Base32, and Base64 Data Encodings (RFC 4648). Internet Engineering Task Force. Recuperado de <https://datatracker.ietf.org/doc/html/rfc4648>
- Mozilla Foundation. (s.f.). Base64 – MDN Web Docs. Recuperado de <https://developer.mozilla.org/en-US/docs/Glossary/Base64>
- Open Web Application Security Project. (2023). Authentication Cheat Sheet. Recuperado de https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
- Python Software Foundation. (s.f.). base64 – Base16, Base32, Base64, Base85 Data Encodings. Recuperado de <https://docs.python.org/3/library/base64.html>
- Grok (Modelo de lenguaje de IA). (2025). Interaccion con el usuario sobre el análisis, ejecución y mejoras sobre código Python propuesto y sobre código Flask propuesto. [Fecha de la interacción: 24 De Mayo De 2025]. <https://grok.com/chat/95a5866a-c861-4a4f-8db9-d2ca343030cf>

Bibliografía

- Google. (2024). Gemini (versión 1.5) [Modelo de lenguaje]. <https://gemini.google.com/>
- OpenAI. (2024). ChatGPT (versión GPT-4) [Modelo de lenguaje]. <https://chat.openai.com/>