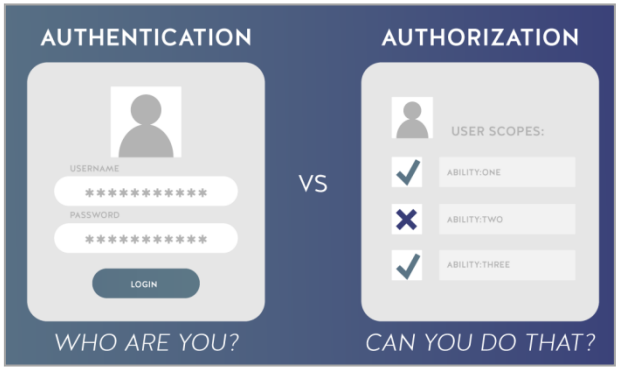
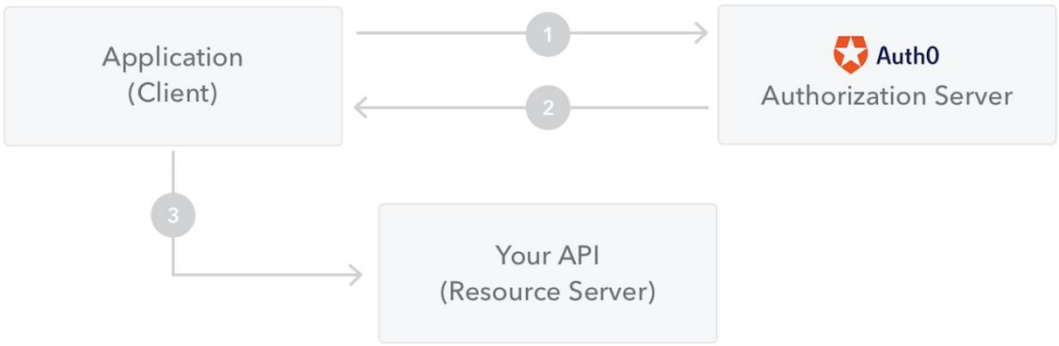
JWT

JSON Web Token (JWT) é comumente utilizado em sistemas de autenticação, tido como uma alternativa mais leve e compacta aos tradicionais *cookies* de sessão.



Os *tokens* JWT precisam ser guardados apenas no lado do cliente e a verificação é feita no lado do servidor através da análise de seu conteúdo e assinatura, sem a necessidade de armazenamento, ao passo que os *cookies* precisam ser armazenados tanto no lado do servidor como no lado do cliente. Por ocuparem menos espaço de armazenamento, comumente são vistos como uma alternativa bastante plausível para autenticação de APIs, especialmente para aplicações *mobile* e IoT.

Observe o funcionamento exemplificado pela figura a seguir. Em um primeiro momento, a **(1)** aplicação cliente envia os dados de usuário e senha para o servidor autenticá-los. Caso a autenticação seja positiva, **(2)** o cliente recebe em seguida o *token*, **(3)** para posteriormente requisitar rotas que estejam protegidas, sempre enviando o *token*:



Fonte: https://jwt.io/introduction

Portanto, JWT é uma alternativa leve, eficiente e compacta de transmitir informações entre sistemas web de forma segura. Definido na RFC 7519, JWT é um objeto JSON que traz consigo uma informação **codificada** e **assinada**, de forma que ela seja confiável entre duas partes.

É importante ressaltar que "codificar" e "assinar" não são a mesma coisa que "encriptar":

> **a)** *codificar uma informação tem como objetivo mudar o seu padrão de representação, porém essa mudança é facilmente reversível.*
>
> **b)** *encriptar uma informação tem como objetivo fazer com que apenas as partes autorizadas consigam acessa-la.*

Nesse sentido, o processo de assinar uma informação tem como objetivo provar a autenticidade da mesma, para garantir que ela tem uma origem conhecida (confiabilidade) e que não foi modificada desde que foi criada (integridade). Em suma, JWT não é encriptado por padrão.

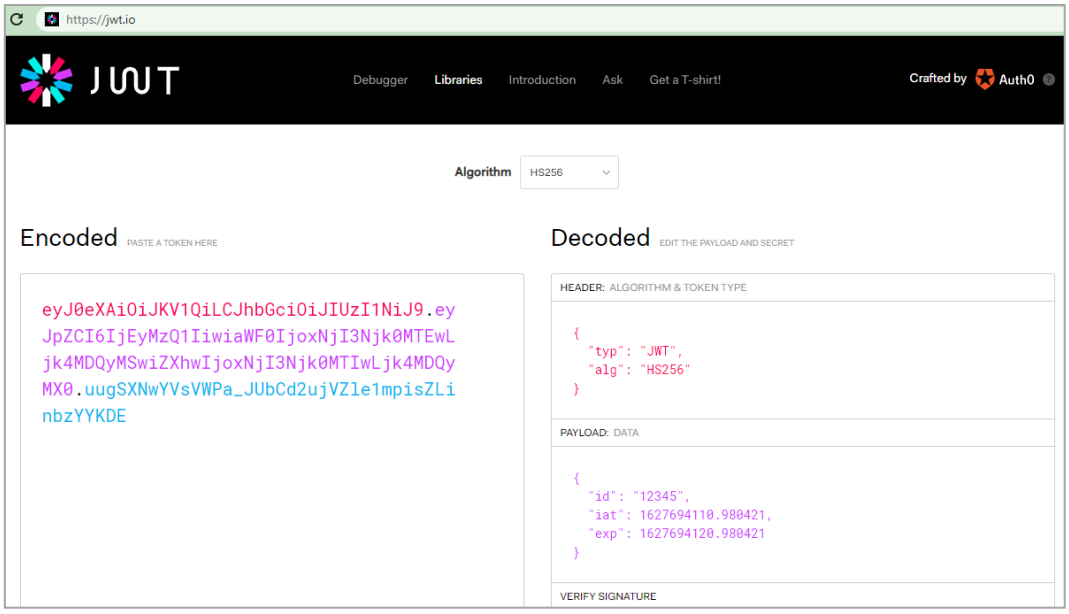
Um *token* JWT é formado por três partes:

Header (cabeçalho)	Payload (dados)	Signature (assinatura)

Veja o exemplo de um token JWT:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjEyMzQ1IiwiaWF0IjoxNjI3Njk0MTEwLjk4MDQyMSwiZXhwIjoxNjI3Njk0MTIwLjk4MDQyMX0.uugSXNwYVsVWPa_JUbCd2ujVZle1mpisZLinbzYYKDE

Acesse o site https://jwt.io e cole a string acima, como na figura:



Perceba que apenas os processos de codificação e assinatura estão presentes no *token*, ou seja, não há encriptação de nenhuma informação. Portanto, o JWT não busca esconder a informação, seu objetivo é garantir a confiabilidade e a integridade dos dados que carrega.

Observe que o cabeçalho (codificado em Base64) consiste em duas partes:

1) *"typ" é o tipo do token, que é JWT,*

2) *"alg" é o algoritmo de assinatura digital que está sendo usado, que neste caso é o HS256 (referência a HMAC com SHA256).*

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Veja o *payload* com o identificador de um cliente fictício (**id**), a data/hora de emissão do *token* (**iat** – que vem de "*Issued At*") e a data/hora de expiração do token (**exp** – que vem "*Expiration Time*"):

PAYLOAD: DATA

```
{
  "id": "12345",
  "iat": 1627694110.980421,
  "exp": 1627694120.980421
}
```

Os dados do *payload* correspondem ao próprio JSON, onde as informações foram inseridas. Cada atributo do *payload* é chamado de *claim* e existem dois tipos:

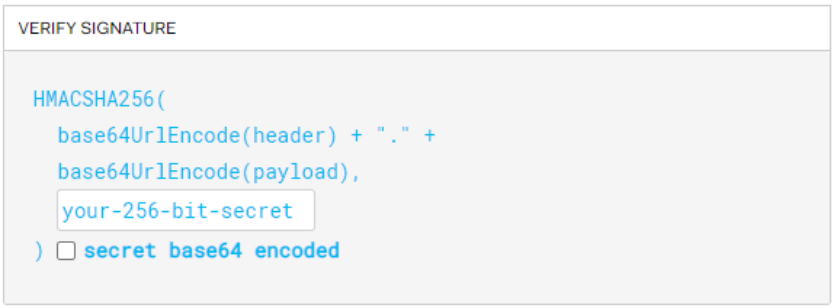
1) ***claims* registradas:** adicionam funcionalidades opcionais ao JWT, como por exemplo, a data de expiração ("exp"), data de emissão ("iat"), etc.

2) ***claims* privadas:** criadas pelo sistema para guardar algum tipo de informação que o desenvolvedor deseja.

Os *claims* dependem das necessidades da aplicação, portanto não há uma "receita de bolo".

Referência: https://www.iana.org/assignments/jwt/jwt.xhtml

Observe a assinatura do token, inclusive a informação de que os dados foram codificados em "Base64" e que existe uma chave secreta de 256 bits:



A assinatura (*signature*) é obtida através do *hash* das informações anteriores, utilizando o algoritmo indicado no campo "alg" do *header*, e depois codificando o resultado para Base64.

Para que o sistema de autenticação da API funcione corretamente, deve-se checar se a assinatura do *payload* e do *header,* contidos no JWT, é a mesma que está no token.

Referências: [1] https://www.base64encode.net/share/Gd9u e [2] https://pt.wikipedia.org/wiki/Base64#:~:text=Base64%20%C3%A9%20um%20m%C3%A9todo%20para,arquivos%20anexos%20por%20e%2Dmail.

O código a seguir foi o responsável por gerar o *token* de exemplo. Faça um teste com ele!

```
import jwt
import datetime

print('----------------------------------------')

# Obtendo a data/hora atual
datahora_emissao = datetime.datetime.now()

# Somando 30 segundos a data/hora atual
datahora_expirar = datahora_emissao + datetime.timedelta(seconds=10)

print('Data/hora de emissao:')
print(datahora_emissao)

print('Data/hora para expirar:')
print(datahora_expirar)

# Convertendo para o formato "timestamp", requerido pelo JWT
datahora_emissao = datahora_emissao.timestamp()
datahora_expirar = datahora_expirar.timestamp()

print('----------------------------------------')

payload = {
            'id': '12345',
            'iat': datahora_emissao,
            'exp': datahora_expirar
          }

print('Meu payload: ')
print(payload)

print('----------------------------------------')

token_encode = jwt.encode(payload, "chavesecreta", algorithm="HS256")
```

```
print('Token codificado: ')
print(token_encode)


print('----------------------------------------')

token_decode = jwt.decode(token_encode, "chavesecreta", algorithms=["HS256"])

print('Token decodificado: ')
print(token_decode)


print('----------------------------------------')

token_datahora_emissao = datetime.datetime.fromtimestamp(token_decode['iat'])
token_datahora_expirar = datetime.datetime.fromtimestamp(token_decode['exp'])

print('Token decodificado - data/hora da emissao:')
print(token_datahora_emissao)

print('Token decodificado - data/hora de expirar:')
print(token_datahora_expirar)
```

Observe a saída quando o script foi executado em 30/07/2021:

```
----------------------------------------
Data/hora de emissao:
2021-07-30 22:15:10.980421
Data/hora para expirar:
2021-07-30 22:15:20.980421
----------------------------------------
Meu payload:
{'id': '12345', 'iat': 1627694110.980421, 'exp': 1627694120.980421}
----------------------------------------
Token codificado:
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjEyMzQ1IiwiaWF0IjoxNjI3Njk0MTEwLjk4MDQyMSwiZXhwIjoxNjI3Njk0MTIwLjk4MDQyMX0.uugSXNwYVsVWPa_JUbCd2ujVZle1mpisZLir
----------------------------------------
Token decodificado:
{'id': '12345', 'iat': 1627694110.980421, 'exp': 1627694120.980421}
----------------------------------------
Token decodificado - data/hora da emissao:
2021-07-30 22:15:10.980421
Token decodificado - data/hora de expirar:
2021-07-30 22:15:20.980421
```

*Obs.: em Python 3x, quando aparecer o prefixo "**b**" antes da variável string que armazena o token, significa "bytes", e neste caso um literal de string.*

MODULARIZANDO O CÓDIGO E TRANSFORMANDO-O EM UMA BIBLIOTECA

O código abaixo foi modularizado e aperfeiçoado. Observe que duas funções foram criadas: **create_token** *e* **verify_token**

Além disso, o *payload* agora possui as informações "*user*", "*iat*" e "*exp*".

Arquivo **"jwt_api.py"** é a biblioteca com as funções de criação e verificação de *tokens* jwt :

```
import jwt
import datetime


def create_token(user, expireminutes, secretkey):
    creation_date = datetime.datetime.now()
    expiration_date = creation_date + datetime.timedelta(minutes=expireminutes)

    creation_date = creation_date.timestamp()
    expiration_date = expiration_date.timestamp()

    payload = {
        'user': user,
        'iat': creation_date,
        'exp': expiration_date
    }

    try:
        token_encode = jwt.encode(payload,
                                  secretkey,
                                  algorithm="HS256")
    except:
        token_encode = None

    return token_encode




def verify token(token, secretkey):

    try:
        dec_token = jwt.decode(token,
                               secretkey,
                               algorithms=["HS256"],
                               options={"verify signature": True})

        if dec_token['exp'] >= datetime.datetime.now().timestamp():
            payload = {
                'user': dec_token['user'],
                'iat': datetime.datetime.fromtimestamp(dec_token['iat']),
                'exp': datetime.datetime.fromtimestamp(dec token['exp'])
            }
        else:
            payload = None
    except:
        payload = None


    return payload
```

Arquivo “**teste_jwt.py**” é um exemplo de como utilizar a biblioteca e proteger suas rotas:

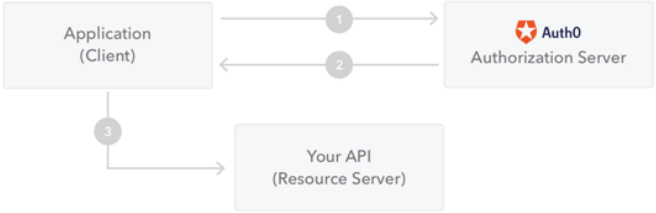
```
from flask import request
from flask import Flask
from flask import jsonify

import jwt_api

app = Flask(__name__)
app.secret_key = 'minha_chave_unica'


@app.route('/auth', methods=['POST'])
def auth():
    token = None
    if request.method == 'POST':
        user = request.json['user']
        password = request.json['password']

        if user == 'elsonabreu' and password == '12345':
            token = jwt_api.create_token(user, 5, app.config['SECRET_KEY'])

    return jsonify({"token": str(token)}),200



@app.route("/protect", methods=['POST'])
def protect():
    if request.method == 'POST':
        token_received = request.json['token']

        payload = jwt_api.verify_token(token_received, app.secret_key)

        if payload != None:
            return jsonify({"msg": "Access route protect " + payload['user']}),200
        else:
            return jsonify({"msg": "Access denied"}),403

    else:
        return jsonify({"msg": "Access denied"}),403



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

O correto uso prevê que:

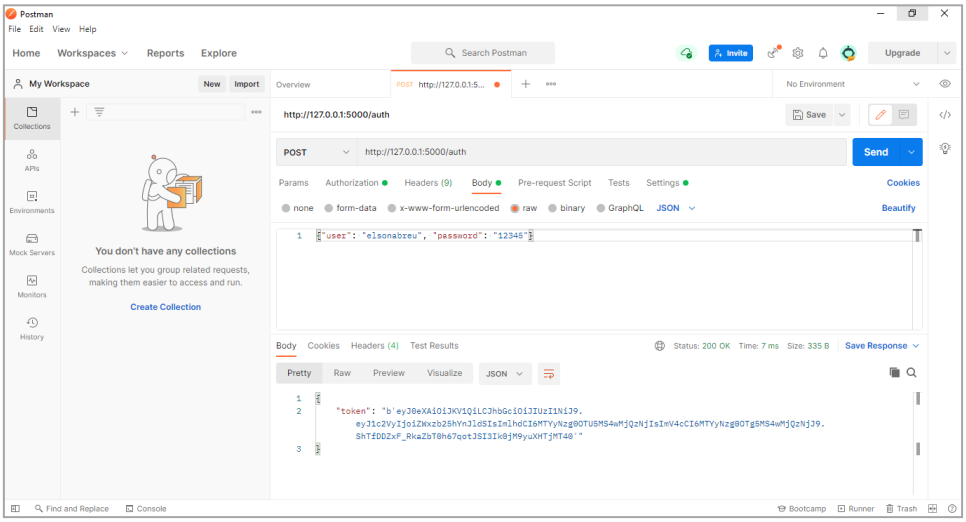
(1) a rota “**/auth**” seja chamada e receba via POST um JSON contendo o ***{ user , password }.***

(2) caso o usuário/senha esteja de acordo, um *token* deverá ser gerado e retornado para aplicação cliente.

(3) o acesso à rota protegida, chamada de “**/protect**”, espera que um *token* válido seja enviado sempre. Justamente o *token* do passo anterior!

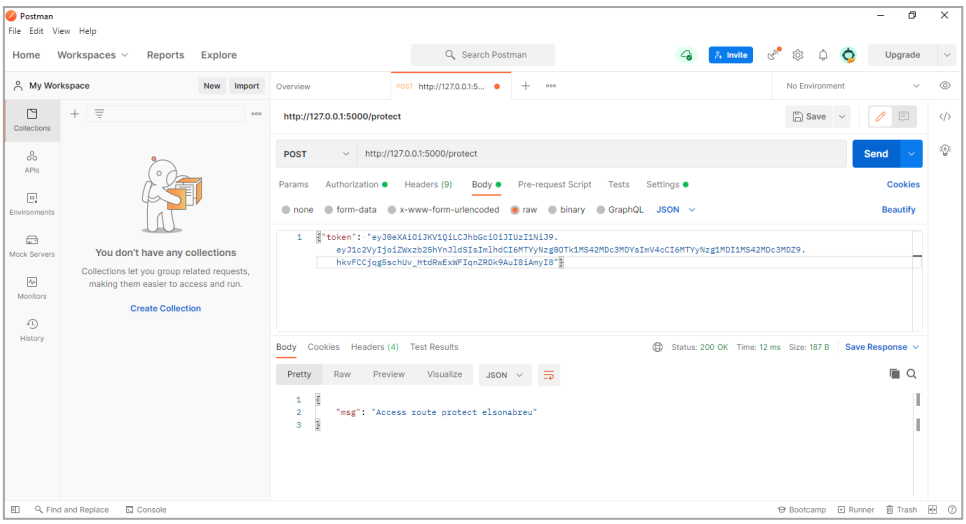


Utilizando o software POSTMAN para testar a API, observe que o *endpoint* usado foi "**HTTP://127.0.0.1/auth**" e o JSON enviado pelo método POST foi **{"user": "elsonabreu", "password": "12345"}**. A resposta foi um *token* JWT contendo as informações de nome de usuário, data de emissão, data para expirar e assinatura (*em formato JSON, claro):*



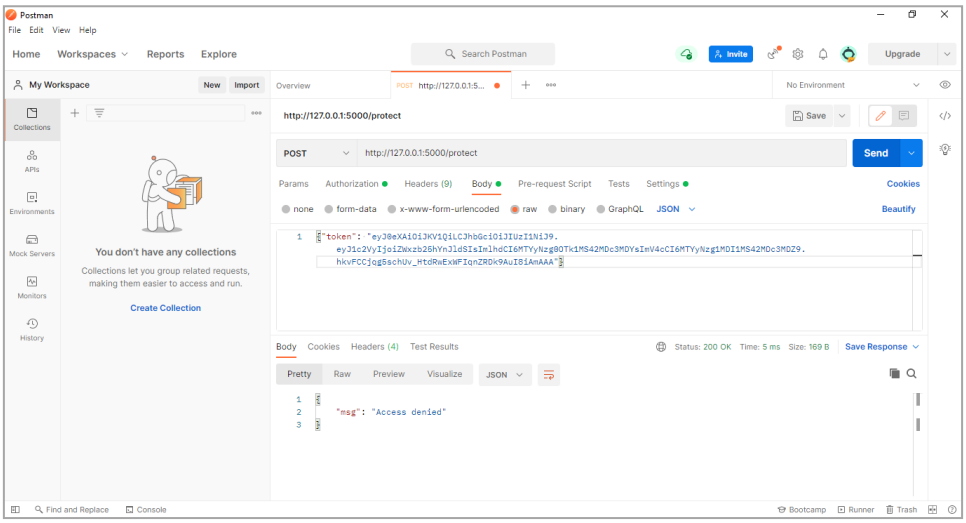
De posse do *token*, mudamos o *endpoint* para "**HTTP://127.0.0.1/protect**" para tentar acessar a rota protegida. *Você pode utilizar a técnica em todas as rotas que precisarem ser protegidas!*

Observe também que o JSON retornado pelo POST é {**"token"**: **"b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiZWxzb25hYnJldSIsImlhdCI6MTYyNzg0OTU5MS4wMjQzNjIsImV4cCI6MTYyNzg0OTg5MS4wMjQzNjI9.ShTfDDZxF_RkaZbT0h67qotJSI3Ik0jM9yuXHTjMT40'"**} :



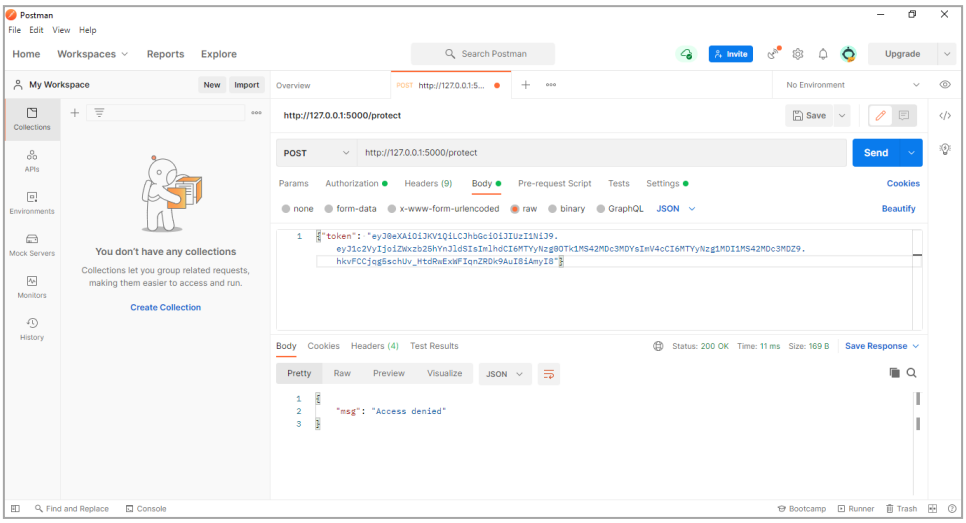
A mensagem de resposta é que a rota protegida foi acessada pelo usuário "elsonabreu".

Mas para ter certeza que o algoritmo funciona, veja que o token teve os três últimos caracteres alterados para "**AAA**", e posteriormente submetido para a rota protegida. Isso faz com que a mensagem de "*Access denied*" seja retornada, sinalizando o acesso negado:



Perceba que a assinatura depende da criptografia dos dados contidos no *payload*, a chave secreta contida no código é o grande trunfo, por isso escolha uma que não seja tão trivial quanto no exemplo passado anteriormente! Isso garante que não possa ser forjado um *token* para acesso não autorizado, a não ser que sua chave seja muito simples e que o intruso consiga descobri-la, além de identificar um nome de usuário válido. Lembre-se que a assinatura é usada para verificar se a mensagem não foi alterada ao longo do caminho e, no caso de *tokens* assinados com uma chave privada, também pode verificar se o remetente do JWT é quem diz ser. **Obs.:** *é preciso retirar o 'b' que antecede a sequencia do token que foi enviada, já que sinaliza que o formato é de bytes (coisas do Python ;-)).*

Em outro teste, observe que o *token* enviado após 5 minutos também recebe a mensagem de acesso negado, mesmo com a assinatura correta! Tal mensagem é devido ao tempo padrão utilizado no algoritmo ter sido excedido (5 minutos):



Mas se você quiser saber como é o código Python 3x para consumir a API, veja abaixo:

```
import requests

endpoint1 = 'http://127.0.0.1:5000/auth'
endpoint2 = 'http://127.0.0.1:5000/protect'


credentials = {"user": "elsonabreu", "password": "12345"}

r = requests.post(url=endpoint1, json=credentials)

if r.status_code == 200:
    # Obtendo token
    token = r.json()
    signature = token['token']

    # Removendo sinalizacao de "bytes" da string (caractere 'b' e apostrofos)
    signature = signature.replace('b\'', '')
    signature = signature.replace('\'', '')


    # Enviando o token para endpoint protegido
    r = requests.post(url=endpoint2, json={'token': signature})

    if r.status_code == 200:
        print('Acesso concedido : ' + r.text )
    else:
        print('Acesso negado : ' + r.text)

else:
    print('Falha na requisicao.')
```

Para complementar o assunto “segurança de APIs” (que é bastante extenso e denso), algumas recomendações dadas pela Red Hat são bastante pertinentes:

- **Usar *tokens*:** estabeleça identidades confiáveis e controle o acesso a serviços e recursos usando *tokens* atribuídos a essas identidades.

- **Usar criptografia e assinaturas:** criptografe os dados usando métodos como o protocolo TLS descrito acima. Exija o uso de assinaturas para garantir que apenas os usuários certos descriptografem e modifiquem os dados.

- **Identificar vulnerabilidades:** monitore os componentes de sistema operacional, rede, *drivers* e APIs. Saiba como tudo isso funciona junto e identifique os pontos fracos que podem ser usados para invadir as APIs. Use *sniffers* para detectar problemas de segurança e rastrear vazamentos de dados.

- **Usar cotas e limites:** estabeleça cotas de chamadas de API e monitore o histórico de uso. Um número elevado de chamadas de uma API pode indicar abuso. Isso também pode significar um erro de programação, como realização de chamadas de API em loop infinito. Crie regras de limitação para proteger as APIs contra ataques de DNS e picos de serviço.

- **Use um *gateway* de API:** *gateways* de API funcionam como o principal ponto de controle do tráfego de API. Um bom *gateway* permite autenticar o tráfego, bem como controlar e analisar o uso das APIs.

Fonte: https://www.redhat.com/pt-br/topics/security/api-security