Sécurisation des APIs

I. Introduction à la sécurité des APIs

Les APIs sont devenues aujourd’hui omniprésentes dans le monde de l’informatique. Elles permettent l’échange de données entre différentes applications. Il est donc légitime de se poser la question sur la sécurisation des données qui sont échangées. Ce cours vise à acquérir les connaissances nécessaires pour ce faire.

**Compétences requises pour ce cours :**

* FastAPI et Flask

Intérêts de la sécurité

Nous pouvons distinguer deux types d’APIs en fonction du type d'accès aux différentes ressources. Nous avons les APIs qui fournissent un accès public à leurs ressources (les données de la météo par exemple) et nous avons également les APIs dont les données sont privées et donc limitées à certains utilisateurs (des données bancaires par exemple). En fonction du niveau d’accès des ressources, les APIs ne nécessiteront pas le même niveau de vigilance.

**Ressources publiques**

Comme leur nom l’indique, les informations contenues dans ces ressources sont publiques.

Par exemple, dans le cadre d'un blog public, on souhaite probablement que le point de terminaison (*endpoint*) GET /posts, qui renvoie les différents articles du blog, soit accessible publiquement, sans aucune authentification préalable de l’utilisateur.

Même si les données exposées sont publiques, il est conseillé de mettre en place une clé API, ce qui vous permettra notamment de savoir quel service effectue la requête, et potentiellement de mettre en place des quotas pour éviter les abus.

**Ressources privées**

Il s’agit de ressources auxquelles tous les utilisateurs ne doivent pas avoir accès. En général, il s'agit d'informations propres à un utilisateur : un historique de commandes, des informations bancaires, des données personnelles, etc. De telles informations ne doivent être accessibles que par leur propriétaire.

Afin de garantir la sécurité des APIs, il faut s'appuyer sur 3 concepts importants : l'authentification, l'autorisation et la traçabilité. L'authentification de l'autorisation sont parfois confondus mais sont pourtant deux notions bien distinctes. Il faut faire attention à bien les différencier.

**Authentification**

L'authentification, c'est savoir quel utilisateur fait appel à notre API. A partir des informations d'authentification transmises (par exemple le nom d'utilisateur et mot de passe), nous pouvons identifier l'utilisateur en question et s'assurer que c'est bien lui qui fait la requête. On peut comparer cette étape à la vérification d'une carte d'identité par un douanier.

**Autorisation**

Une fois que l'utilisateur est authentifié et identifié par l'API, l'autorisation sert à savoir ce que cette personne a le droit de faire ou non. Durant chaque appel HTTP de la part de l'utilisateur, il faut vérifier s'il a bien droit d'accéder à la ressource en question. Dans la continuité de la comparaison, le douanier vérifie dans son fichier que la personne a bien le droit d'entrer sur le territoire.

**Traçabilité**

La traçabilité permet de savoir qui a fait quoi, quand et avec quelles ressources. On peut par exemple simplement sauvegarder ces informations dans un simple fichier de logs (appelé aussi journaux, qui est un fichier où se trouvent différentes informations sur les échanges de données etc).

Afin de garantir la traçabilité de l'API, il faut sauvegarder pour chaque appel HTTP les informations suivantes :

* **Quoi**
  + La ressource concernée (par exemple /users)
  + La méthode HTTP appelée sur cette ressource (GET, POST, DELETE, etc.)
* **Qui**
  + L’utilisateur final à l’origine de l’appel
  + L’application qu’utilise cet utilisateur : une API peut être utilisée par plusieurs applications différentes (application mobile ou site web par exemple)
  + L'adresse IP source
* **Quand**
  + Le moment où a été exécutée la requête (date et heure)

Une telle traçabilité est par exemple nécessaire pour pouvoir effectuer des audits de sécurité, ou pour determiner la provenance d'une attaque si l'on découvre que l'API a été compromise.

On pourra rajouter à cela des informations telles que le type d’appareil utilisé ou la provenance (approximative) de la requête (grâce à des méthodes simples de géolocalisation).

À partir des définitions que nous avons données jusque-là, on peut déduire que pour une ressource privée, il nous faudra gérer l’Authentification et l’Autorisation. Enfin, la traçabilité sera nécessaire pour l'ensemble des APIs et des ressources, publiques ou privées.

**Principales méthodes**

Afin de garantir la sécurité des APIs, il existe plusieurs méthodes plus ou moins complexes, et plus ou moins sécurisées. Il faudra choisir la méthode la plus adaptée en fonction des spécifications et du type d'API.

Parmi les différentes méthodes, nous présenterons notamment les **clés API**, **HTTP Basic Auth**, les JSON Web Tokens ou **JWT**.

Nous présenterons également le protocole **HTTPS**, qui est une extension du protocole HTTP qui permet d'encrypter les données échangées entre le client et le serveur.

## II. Les clés API (API Keys)

Une clé API (ou API key) est un identifiant unique utilisé pour authentifier un utilisateur ou programme auprès d'une API. C'est une chaîne de caractère que l'on peut inclure soit dans l'URL de la requête, soit en en-tête (header) de la requête.

Les clés API peuvent notamment être utilisées pour :

* Identifier l'émetteur (utilisateur ou programme) de la requête
* Appliquer des quotas (par exemple pour limiter le nombre de requête par mois)
* Facturer l'utilisation de l'API
* Appliquer des niveaux d'autorisations

Les clés API sont notamment très utilisées pour les APIs publiques.

### **Implémentation avec FastAPI**

Créez un fichier api.py dans lequel vous mettrez le code suivant :

from fastapi import Security, Depends, FastAPI, HTTPException, status

from fastapi.security.api\_key import APIKeyHeader, APIKey

API\_KEY = "1234567asdfgh"

API\_KEY\_NAME = "access\_token"

api\_key\_header = APIKeyHeader(name=API\_KEY\_NAME, auto\_error=False)

api = FastAPI()

Depuis le module fastapi, nous importons les classes et modules pour gérer les dépendances (Depends), la sécurité (Security), les exceptions (HTTPException) et les status HTTP (status).

Depuis le module fastapi.security.api\_key, nous importons les classes APIKeyHeader et APIKey. Nous instancions ensuite notre clé API (api\_key\_header) en lui donnant un nom.

L'attribut auto\_error passé lors de l'instanciation indique si l'on gère nous même les erreurs d'authentification (False) ou si FastAPI le gère automatiquement (True). Dans cet exemple, nous gérons nous même ces erreurs.

Nous allons donc implémenter la fonction qui vérifie si la clé reçue correspond bien à la clé API que nous avons définie. Rajoutez le code ci-dessous :

async def get\_api\_key(api\_key\_header: str = Security(api\_key\_header)):

if api\_key\_header == API\_KEY:

return api\_key\_header

raise HTTPException(

status\_code=status.HTTP\_403\_FORBIDDEN,

detail="Could not validate credentials"

)

On implémente une fonction get\_api\_key qui récupère la clé API reçue en en-tête (header) de la requête.

Déclarer l'argument api\_key\_header: str = Security(api\_key\_header) permet de déclarer une dépendance. Cette dépendance exige une clé API pour accéder à cette fonction. Dans le corps de cette fonction on vérifie si la clé obtenue correspond à note clé API. Si c'est le cas, on renvoie la clé, sinon on lève une exception HTTPException correspondant au code HTTP 403 (Forbidden).

Nous allons maintenant implémenter deux routes : une sans clé requise et une avec une clé requise. Ajoutez le code ci-dessous :

@api.get("/hello")

def hello():

return 'Hello World !'

@api.get("/secure")

async def secure(api\_key\_header: APIKey = Depends(get\_api\_key)):

return 'Hello this is secure ! '

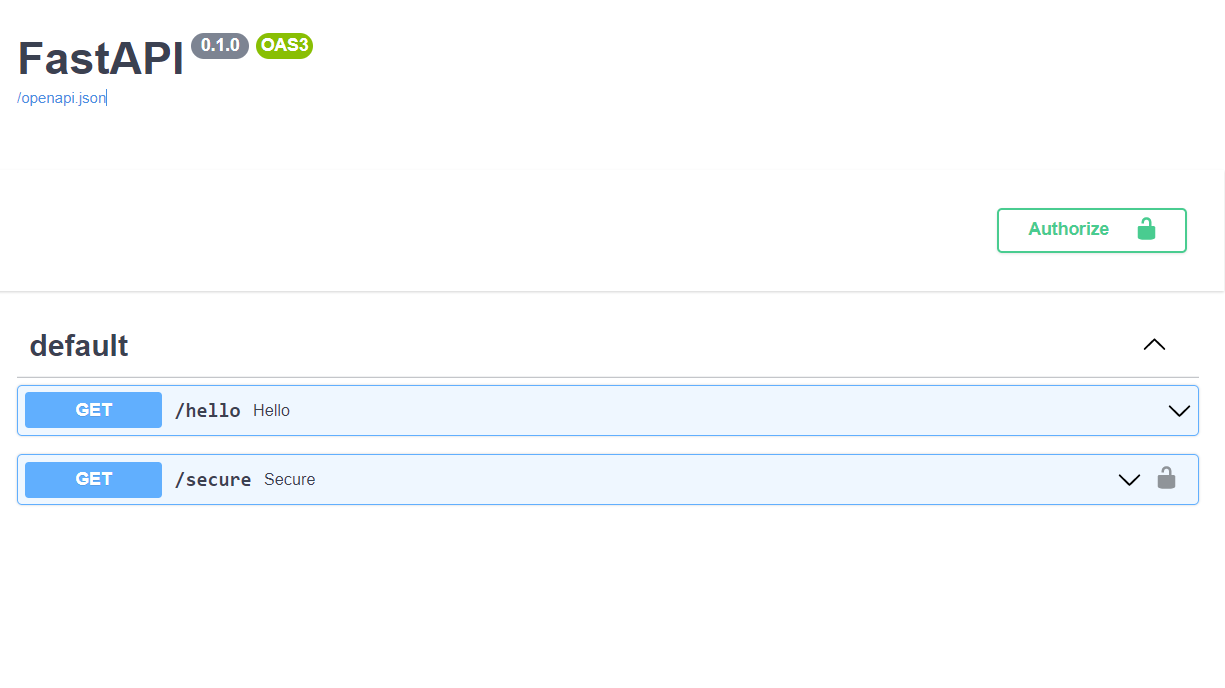
Nous avons défini deux points de terminaisons (endpoint) :

* Le endpoint GET /hello qui ne requiert pas de clé API.
* Le endpoint GET /secure qui requiert une clé API. On définit en argument dans la fonction secure une dépendance à la fonction get\_api\_key à l'aide de Depends. Ce qui veut dire que pour accéder à cette route, il faut recevoir au préalable une clé API valide.

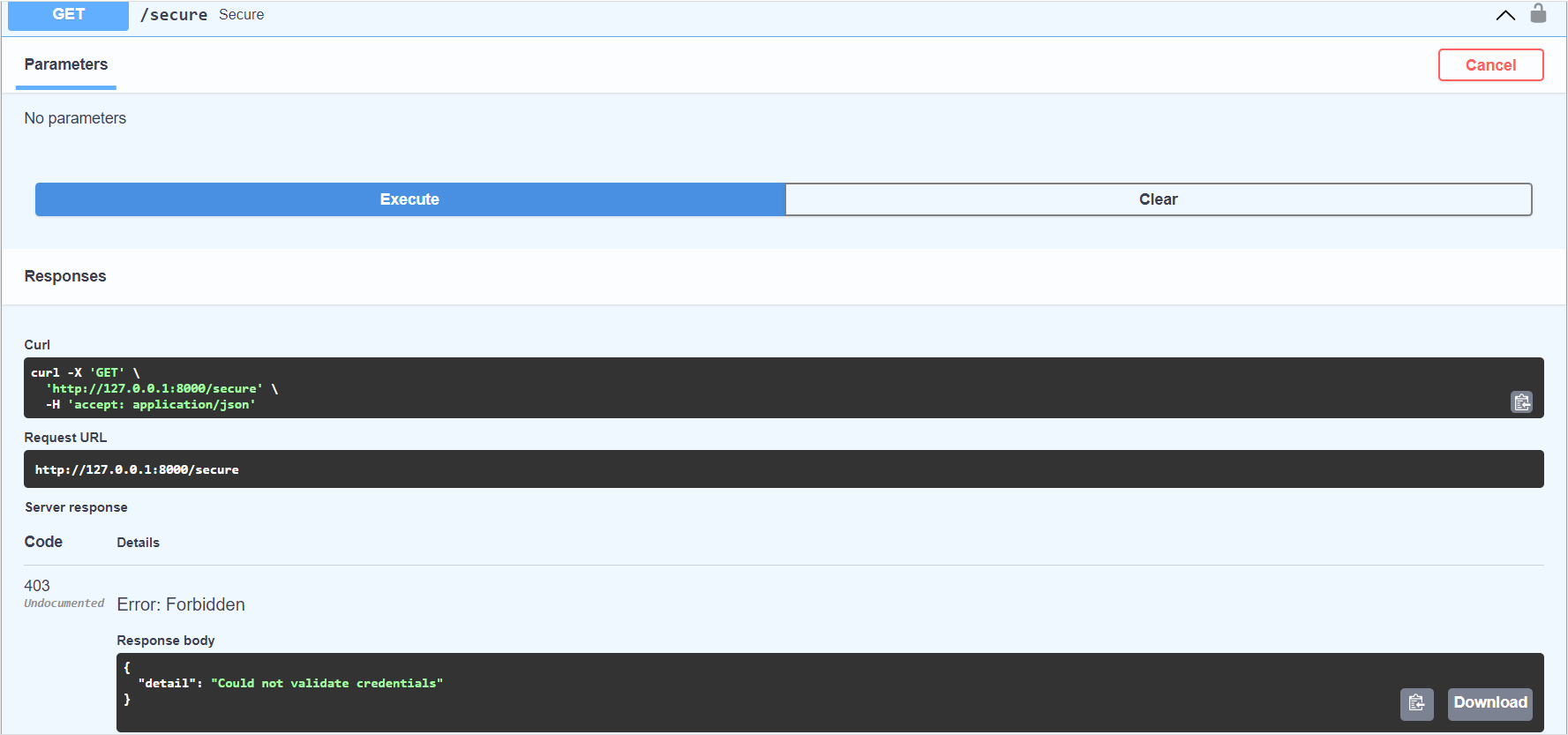
Sur le terminal, lancez l'API à l'aide de la commande :

uvicorn api:api --reload

En allant sur la documentation http://localhost:8000/docs (en local) ou http://ip\_machine\_virtuelle:8000/docs (si vous utilisez la machine virtuelle), nous obtenons le résultat suivant :



En exécutant la route secure, nous obtenons le résultat suivant :



Nous obtenons l'erreur suivante :

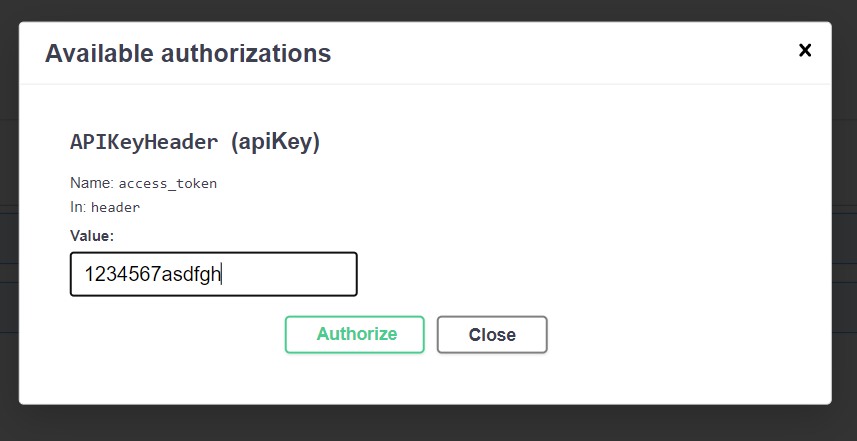
{

"detail": "Could not validate credentials"

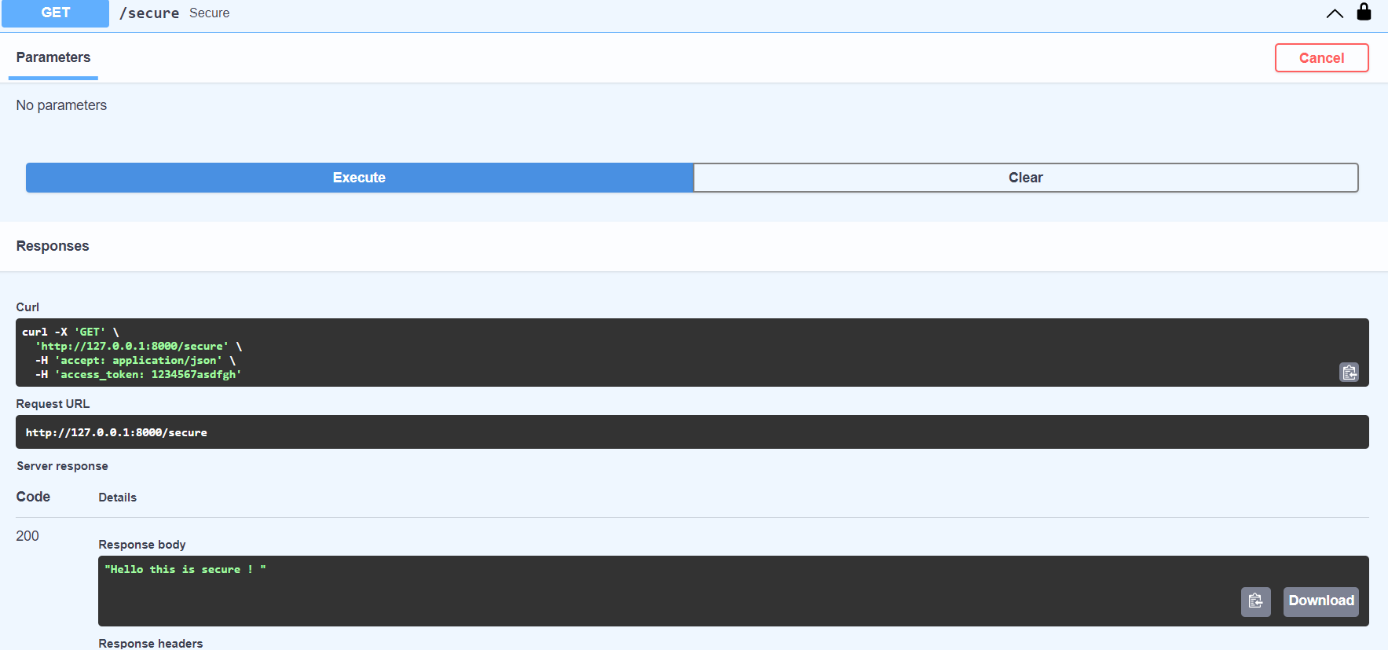
}

Nous avons une erreur 403. En effet, nous ne nous sommes pas authentifiés à l'aide de notre clé API.

En appuyant sur le bouton Authorize, entrez la clé API (1234567asdfgh) comme dans l'image suivante :



Nous pouvons maintenant accéder à la route secure :



La clé API se trouve dans l'en-tête de la requête comme suit :

curl -X 'GET' 'http://127.0.0.1:8000/secure' -H 'access\_token: 1234567asdfgh'

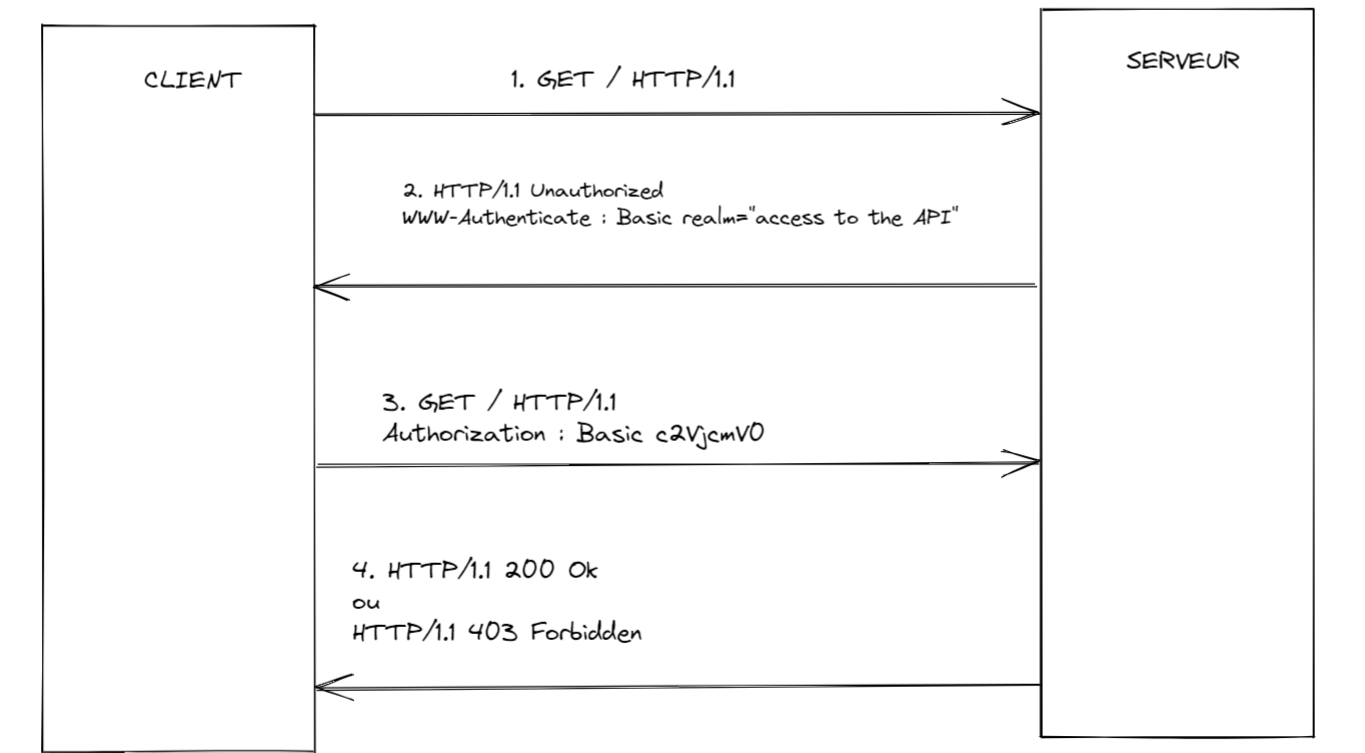
Nous avons donc vu comment utiliser une clé API avec FastAPI. Ces clés peuvent être utiles pour des accès à des ressources publiques. Cependant, pour des ressources privées, il est recommandé d'utiliser d'autres méthodes d'authentification (que nous verrons lors des prochains chapitres) pour assurer une sécurité accrue.

## III. HTTP Basic Auth

L'authentification basique HTTP est une méthode simple permettant au serveur de demander des informations d'authentifications au client, et à ce dernier de les fournir.

Le client transmet les informations d'authentification au serveur dans un en-tête d'autorisation. Les informations d'authentification sont encodées en base-64. Plus précisément, c'est la chaîne de caractère username:password qui est encodée en base 64 puis transmise dans l'en-tête (header) de la requête.

Le processus d'authentification HTTP est défini à l'aide du schéma suivant :



Ce processus est composé de quatre étapes :

1. Le client envoie une requête au serveur
2. Le serveur envoie une réponse HTTP avec un code d'état 401 (Non autorisé) et un en-tête WWW-Authenticate. La plupart des clients Web gèrent cette réponse en demandant un identifiant et un mot de passe à l'utilisateur.
3. Le client envoie ces identifiants encodés en base64 au serveur via l'en-tête Authorization.
4. Le serveur vérifie les identifiants de l'utilisateur. Si ils sont valides et que l'utilisateur a accès à la ressource, le serveur pourra renvoyer la réponse attendue par le client. Si les identifiants ne sont pas valides, une erreur 401 sera renvoyée. Enfin, si les identifiants sont valides mais que l'utilisateur n'a pas les droits pour accéder à la ressource en question, une erreur 403 sera renvoyée.

### **Implémentation avec Flask**

Dans cette partie, nous allons voir comment implémenter la méthode HTTP Basic Auth avec Flask. Pour cela, nous allons devoir installer le module qui fournit cette méthode avec Flask.

Exécutez cette commande sur le terminal

pip install flask\_httpauth

Dans un fichier nommé flask\_http\_basic.py, mettez le code suivant :

from flask import Flask

from flask\_httpauth import HTTPBasicAuth

from werkzeug.security import generate\_password\_hash, check\_password\_hash

Nous importons d'abord Flask. Ensuite, on importe la méthode HTTPBasicAuth du module flask\_httpauth. Enfin, nous importons les méthodes generate\_password\_hash et check\_password\_hash du module werkzeug.security qui permettent respectivement de créer un mot de passe chiffré à partir d'un mot de passe en clair et de vérifier si un mot de passe en clair correspond bien à un mot de passe chiffré.

Ajoutez le code suivant au fichier

api = Flask(import\_name='my\_api')

auth = HTTPBasicAuth()

users = {

"daniel": {

'password' : generate\_password\_hash("datascientest"),

'private' : 'Private Resource Daniel',

'role' : ['admin', 'user']

},

"john": {

'password' : generate\_password\_hash("secret"),

'private' : 'Private Resource John',

'role' : 'user'

}

}

On instancie notre API et la méthode HTTP Basic Auth à l'aide de la méthode HTTPBasicAuth().

Un dictionnaire users contient pour chaque utilisateur son username, que son password (haché), une ressource privée stockée dans la clé private et enfin les différents rôles qu'il peut avoir dans notre API.

Ajoutez le code suivant

@auth.verify\_password

def verify\_password(username, password):

if username in users and check\_password\_hash(users.get(username)['password'], password):

return username

@auth.get\_user\_roles

def get\_user\_roles(user):

return users.get(user)['role']

HTTPBasicAuth fournit le décorateur verify\_password qui reçoit le nom d'utilisateur et le mot de passe envoyés par le client. Si les informations d'identification sont valides, la fonction renvoie le nom d'utilisateur. Sinon, la fonction peut retourner Noneou False.

Flask-HTTPauth inclut un système d'autorisation simple basé sur des rôles qui peut être ajouté pour filtrer l'accès aux routes. Pour activer la prise en charge des rôles, il faut utiliser le décorateur get\_user\_roles. Nous écrivons ensuite une fonction qui renvoie la liste des rôles pour un utilisateur donné que nous récupérons dans la base de données users.

A présent nous allons définir nos différentes routes qui nécessitent une authentification et qui seront limités à certains rôles.

Ajoutez ces routes

@api.route('/admin')

@auth.login\_required(role='admin')

def admin():

return "Hello {}, vous êtes admin!".format(auth.current\_user())

@api.route('/')

@auth.login\_required(role='user')

def index():

return "Hello, {}!".format(auth.current\_user())

@api.route('/private')

@auth.login\_required(role='user')

def private():

return "Resource : {}".format(users[auth.current\_user()]['private'])

Nous définissons d'abord une route admin, dans laquelle nous utilisons le décorateur login\_required qui impose une authentification et dans lequel nous ajoutons une restriction à cette route en la limitant aux utilisateurs qui sont admin. La fonction admin renvoie un message avec le username de l'utilisateur courant que nous récupérons à l'aide de la méthode current\_user().

Nous faisons de même pour les routes / et /privatequi seront limitées au rôle user et qui renvoient respectivement les informations sur le username de l'utilisateur courant et la ressource privée de celui-ci.

Enfin ajoutez ces lignes de code pour lancer notre API

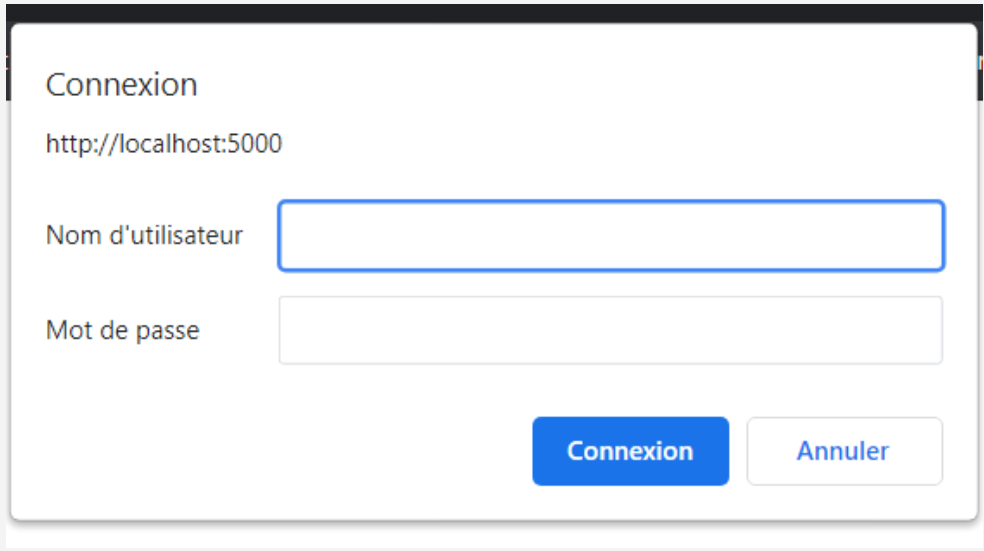
if \_\_name\_\_ == '\_\_main\_\_':

api.run()

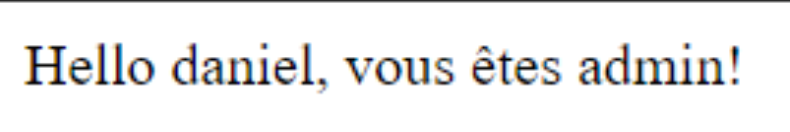
Lancez en ligne de commande l'API en exécutant cette commande :

python flask\_http\_basic.py

En allant sur http://localhost:5000/admin (en local) ou http://ip\_vm:5000/admin (si vous travaillez sur la machine virtuelle), on obtient la fenêtre suivante :



Il faut entrer les identifiants liés à un utilisateur admin (Ici daniel et datascientest). Une fois connecté, nous avons le message retourné par la fonction.



L'utilisateur daniel a tous les rôles, il peut donc accéder à toutes les routes définies. L'utilisateur john, quant à lui, n'aura pas accès à la route /admin.

Si l'on s'est authentifié une première fois en tant que daniel, le navigateur met dans le cache les informations d'authentification, et on ne pourra donc pas essayer de s'authentifier en tant que john pour vérifier qu'il n'a pas accès à la route /admin. Si l'on souhaite s'authentifier à nouveau avec l'utilisateur john, il faudra donc ouvrir une nouvelle page localhost:5000 en navigation privée.

### **Implémentation avec FastAPI**

FastAPI fournit dans son package fastapi.security la méthode HTTP Basic Auth.

Exécutez la commande suivante : pip install passlib[bcrypt]. Ce module nous fournit des fonctions pour hacher des mots de passe.

Dans un fichier nommé fastapi\_http\_basic.py, insérez le code suivant :

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import HTTPBasic, HTTPBasicCredentials

from passlib.context import CryptContext

Nous utilisons les fonctions Depends, FastAPI, HTTPException et status du module fastapi. Depends sera utile pour appliquer des dépendances pour exiger une authentification avant d'accéder à une route. HTTPException et status seront utiles pour lever des exceptions avec des erreurs. Nous importons également HTTPBasicet HTTPBasicCredentialsdu module fastapi.securitypour employer la méthode HTTP Basic Auth et pour utiliser le formulaire d'authentification à l'aide deHTTPBasicCredentials. Enfin, nous avons importé les modules passlib.context pour crypter les mots de passes qui seront donnés dans le formulaire d'authentification.

Rajoutez le code suivant :

app = FastAPI()

security = HTTPBasic()

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

users = {

"daniel": {

"username": "daniel",

"name": "Daniel Datascientest",

"hashed\_password": pwd\_context.hash('datascientest'),

},

"john" : {

"username" : "john",

"name" : "John Datascientest",

"hashed\_password" : pwd\_context.hash('secret'),

}

}

On instancie notre API à l'aide de la méthode FastAPI() , la méthode HTTP à l'aide HTTPBasic() et la variable pwd\_context pour crypter les mots de passe.

Enfin, nous utilisons notre base de données users. Le cryptage des deux mots de passe des utilisateurs correspond au cryptage du mot de passe secret.

Ajoutez cette fonction qui nous permettra de fournir un formulaire d'authentification et de vérification d'identifiants.

def get\_current\_user(credentials: HTTPBasicCredentials = Depends(security)):

username = credentials.username

if not(users.get(username)) or not(pwd\_context.verify(credentials.password, users[username]['hashed\_password'])):

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Incorrect email or password",

headers={"WWW-Authenticate": "Basic"},

)

return credentials.username

Dans le code ci-dessus, on récupère dans la variable credentials les identifiants entrés par l'utilisateur à l'aide de la dépendance Depends(security). Ce qui veut dire que l'on exige par cette dépendance, l'authentification de l'utilisateur à l'aide de la méthode HTTP.

On récupère l'identifiant et le mot de passe de l'utilisateur grâce aux attributs username et password de la variable credentials. On vérifie si l'identifiant est présent dans la base de données. Ensuite, on compare si le mot de passe crypté correspond bien à celui de la base de données en utilisant la méthode verify de la variable pwd\_context. On lève une erreur 401 si ils ne correspondent pas. Sinon, on renvoie l'identifiant de l'utilisateur.

Ajoutez cette route dans le fichier

@app.get("/user")

def current\_user(username: str = Depends(get\_current\_user)):

return "Hello {}".format(username)

Enfin, on définit une route /user ayant la méthode GET. On fait dépendre cette route de la fonction définie précédemment. Ce qui veut dire que pour accéder à cette route, il faut au préalable que l'utilisateur se soit authentifié.

Lancez à présent l'API à l'aide de la commande suivante :

uvicorn fastapi\_http\_basic:app --reload

Allez sur http://localhost:8000/user ou http://ipmachinevirtuelle:8000/user et entrez les identifiants de l'utilisateur :

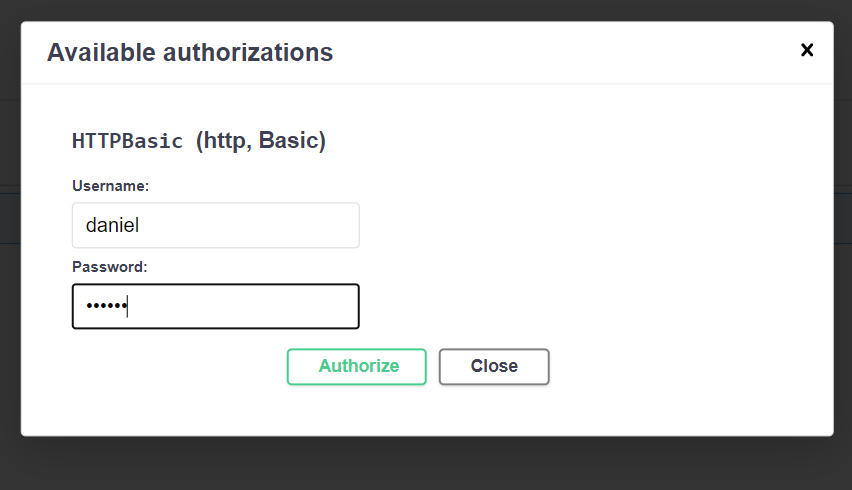
username : daniel

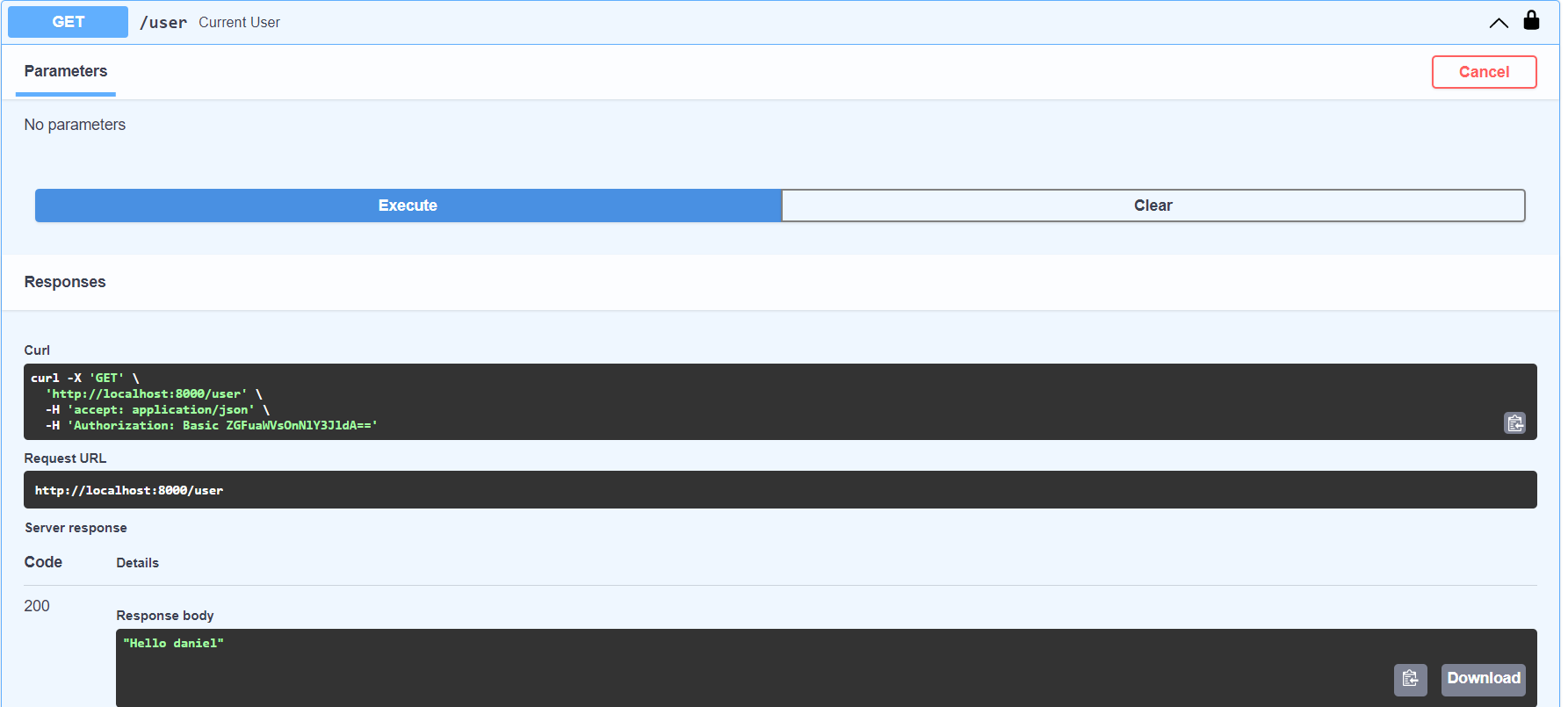
mot de passe : secret

On obtient le résultat suivant :



On peut également tester la route /uservia la documentation en allant sur http://localhost:8000/docs ou http://ipmachinevirtuelle:8000/docs, entrer les identifiants en cliquant sur le bouton Authorize et exécuter la fonction current\_user.

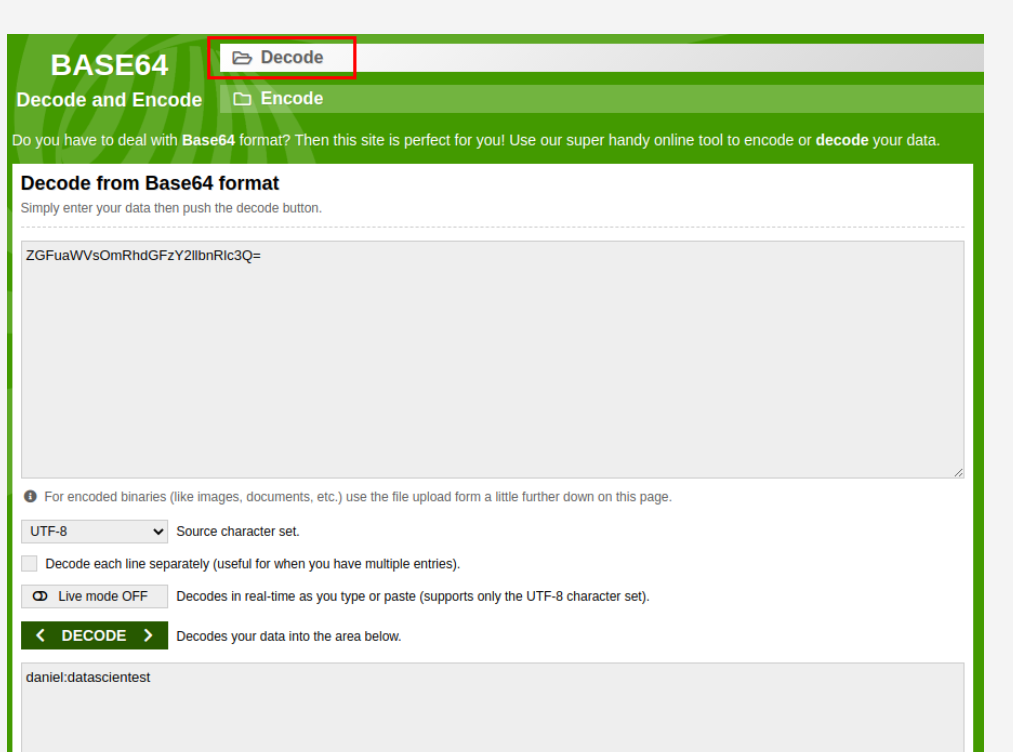




On obtient bien le résultat voulu. Si les identifiants ne sont pas valides, l'API renvoie une erreur.

La méthode HTTP Basic Auth peut présenter des inconvénients lorsqu'elle est utilisée seule. En effet, les identifiants étant encodés en base64 et non haché, ils peuvent être décodés facilement si ils sont interceptés par un attaquant. On peut le voir facilement en utilisant un exemple sur ce site (https://www.base64encode.org/). Si nous prenons comme exemple l'utilisateur daniel sur le site, nous devrons encoder les identifiants daniel:datascientest en base64 comme le montre l'image suivante :

En décodant la chaîne de caractère générée (ZGFuaWVsOmRhdGFzY2llbnRlc3Q=), on retrouve bien les identifiants :



N'importe qui peut donc retrouver les identifiants à partir du header d'autorisation. Pour empêcher cela et rendre la méthode plus sécurisée, il serait judicieux d'utiliser le protocole HTTPS (que l'on définira dans les chapitres suivants) qui permet de sécuriser les échanges entre le client et le serveur en cryptant les messages échangés.

## IV. Les JSON Web Token (JWT)

JSON Web Token (JWT) est une norme ouverte ([RFC 7519](https://tools.ietf.org/html/rfc7519)) qui définit un moyen compact et autonome de transmettre en toute sécurité des informations entre plusieurs parties en tant qu'objet JSON.

Ces informations peuvent être vérifiées car elles sont signées numériquement. Les JWTs peuvent être signés à l'aide de plusieurs algorithmes comme **HS256**, **RSA** ou **ECDSA**.

Lorsqu'un serveur reçoit un JWT, il peut garantir que les données qu'il contient sont dignes de confiance car elles sont signées par la source. Aucun intermédiaire ne peut modifier un JWT une fois qu'il est envoyé.

Il est important de noter qu'un JWT garantit la propriété des données mais pas le cryptage. Les données JSON que vous stockez dans un JWT peuvent être vues par toute personne qui intercepte le jeton car par défaut il est simplement sérialisé et non crypté.

Pour cette raison, il est fortement recommandé d'utiliser HTTPS avec les JWT (que nous verrons dans le prochain cours).

Un JWT, une fois généré par le serveur, est composé de trois parties séparées par des points :

* **Header** : C'est un objet de type JSON (comme un dictionnaire en Python) encodé en base64 et qui représente l'en-tête du token. Le header est composé de deux types : le type de token (ici JWT) et l'algorithme utilisé pour chiffrer le token (En général l'algorithme de hachage utilisé est HS256)
* **Payload** : Le payload est le contenu du token. C'est également un objet JSON qui sera encodé en base64. Il peut être composé de plusieurs types d'informations. On peut trouver :
  + **"sub"** : l'identifiant de l'utilisateur
  + **"aud"** : Il s'agit du client qui a demandé la ressource
  + **"exp"** : La durée de validité du token
  + **"iat"** : La date où le token a été édité
  + **"iss"** : Il s'agit du serveur qui a émis le token

La liste n'est pas complète, il existe d'autres propriétés.

* **Signature** : La signature est l'application de l'algorithme de hachage des deux premières parties citées ci-dessus. L'algorithme utilise une clé secrète (que seul le serveur détient), ce qui veut dire que si une personne modifie le contenu du token, la signature ne sera pas correcte et le token sera ainsi rejeté.

Donnons un exemple pour illustrer ces propos. Considérons un JWT avec le header suivant :

{

"alg": "HS256",

"typ": "JWT"

}

Le header indique que l'algorithme de hachage utilisé est l'algorithme HS256 et que le jeton est un jeton JWT. Une fois encodé en base64, nous obtenons la chaîne de caractère suivante :

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Ensuite nous avons la partie payload :

{

"sub": "1234567890",

"name": "Daniel",

"iat": "1516239022"

}

Nous avons donc le nom du client, l'identifiant de l'utilisateur ainsi que la date d'émission du token. Une fois encodé en base64, nous obtenons la chaîne suivante :

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkRhbmllbCIsImlhdCI6MTUxNjIzOTAyMn0

Nous avons enfin la troisième partie qui est la signature (qui est le hachage de notre header et payload) :

eSaq3mzEMC8zHO49GZkxyHxf-DyKhYXz5Tx\_HevmL8Y

Le JWT obtenu est donc la chaîne suivante (le header, le payload et la signature séparés par un '.') :

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkRhbmllbCIsImlhdCI6MTUxNjIzOTAyMn0.eSaq3mzEMC8zHO49GZkxyHxf-DyKhYXz5Tx\_HevmL8Y

Les jetons JWT sont échangés lors de requêtes HTTP. La méthode la plus courante pour envoyer ces jetons est de les envoyer à travers l’en-tête HTTP Authorization en tant que Bearer token. L'authentification Bearer (également appelée authentification par jeton) est un schéma d'authentification HTTP qui implique des jetons de sécurité appelés token Bearer. Le nom "authentification au porteur" (Bearer authentication) peut être compris comme "donner l'accès au porteur de ce jeton". Le jeton de porteur est une chaîne cryptique, généralement générée par le serveur en réponse à une demande de connexion. Le client doit envoyer ce jeton dans l'en-tête Authorization lorsqu'il effectue des requêtes vers des ressources protégées.

Les JWT est donc une méthode facile et simple d'utilisation pour authentifier les utilisateurs. De plus, cette méthode est également fiable en termes de sécurité. En effet, il est aisé de vérifier qu'un JWT a été compromis.

Nous allons à présent voir comment implémenter la sécurisation des APIs Flask à l'aide des jetons JWT.

### **Mise en place avec Flask**

Flask dispose d'un module nommé flask\_jwt\_extended permettant de gérer les jetons JWT afin de protéger l'accès à nos APIs.

On va d'abord installer ce module.

Lancez la commande suivante

pip install flask\_jwt\_extended==4.2.1

Créez un fichier api\_jwt.py dans lequel vous ajouterez le code suivant :

from flask import Flask

from flask import jsonify

from flask import request

from datetime import timedelta

from flask\_jwt\_extended import create\_access\_token, get\_jwt\_identity, jwt\_required, JWTManager

from passlib.context import CryptContext

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

users\_db = {

"danieldatascientest": {

"username": "danieldatascientest",

"name": "Daniel Datascientest",

"email": "daniel@datascientest.com",

"hashed\_password": pwd\_context.hash('datascientest'),

"resource" : "Module DE",

},

"johndatascientest" : {

"username" : "johndatascientest",

"name" : "John Datascientest",

"email" : "john@datascientest.com",

"hashed\_password" : pwd\_context.hash('secret'),

'resource' : 'Module DS',

}

}

api = Flask(import\_name="my\_api")

api.config["JWT\_SECRET\_KEY"] = "760257bc03d0f18b99ba31e6b90f6b95f368ec67cebae4edd95f22cf04695185" # Change this!

api.config["JWT\_ACCESS\_TOKEN\_EXPIRES"] = timedelta(minutes=30)

jwt = JWTManager(api)

Nous utilisons tous les imports nécessaires pour notre code avec la même base de données que l'exercice précédent mais avec quelques attributs en plus. Nous pouvons accéder à la configuration de notre API Flask à l'aide de l'attribut config. Pour vérifier nos jetons JWT, nous devons utiliser un algorithme et une clé secrète qui permettra de signer le jeton. Pour générer notre clé secrète, nous allons utiliser la commande suivante :

openssl rand -hex 32

Dans notre code, on précise la clé secrète (que nous venons de générer) pour encoder nos jetons JWT en précisant l'attribut JWT\_SECRET\_KEY ainsi que leur durée de validité en précisant la valeur dans l'attribut JWT\_ACCESS\_TOKEN\_EXPIRES.

Enfin, on relie notre API Flask (qui contient les paramètres des JWT) à l'extension flask\_jwt\_extended à l'aide de JWTManager.

Ajoutez le code suivant qui contient les fonctions utilitaires similaires à l'exercice précédent

def check\_password(plain\_password, hashed\_password):

return pwd\_context.verify(plain\_password, hashed\_password)

def get\_user(database, username):

if username in database:

user\_dict = database[username]

return user\_dict

On va ajouter une route login à notre API pour créer notre access\_token

@api.route("/login", methods=["POST"])

def login():

username = request.json.get("username", None)

password = request.json.get("password", None)

user = get\_user(users\_db, username)

if not user or not check\_password(password, user['hashed\_password']):

return jsonify({"msg": "Bad username or password"}), 401

access\_token = create\_access\_token(identity=username)

return jsonify(access\_token=access\_token)

On récupère les identifiants de l'utilisateur. S'ils sont valides, on lui retourne un access\_token à l'aide de la fonction create\_access\_token qui prend en argument l'identité de l'appelant (username dans notre cas).

On définit deux autres routes, mais cette fois-ci protégées. Ajoutez le code suivant :

@api.route("/user", methods=["GET"])

@jwt\_required()

def get\_current\_user():

current\_user = get\_jwt\_identity()

return jsonify(logged\_in\_as=current\_user), 200

@api.route("/resource", methods=["GET"])

@jwt\_required()

def get\_resource():

current\_username = get\_jwt\_identity()

return jsonify({"resource" : get\_user(users\_db, current\_username)['resource'],

"owner": current\_username})

Le décorateur @jwt\_required indique que pour accéder à cette route, il faut un jeton JWT valide. La méthode get\_jwt\_identity renvoie l'identité de l'utilisateur courant.

Ajoutez ce code

if \_\_name\_\_ == "\_\_main\_\_":

api.run()

On va maintenant tester notre code pour mieux comprendre comment cela marche.

Lancez sur un terminal cette commande

python api\_jwt.py

Dans un autre terminal, lancez cette commande

curl -X POST -H "Content-Type: application/json" -d '{"username":"danieldatascientest", "password":"secret"}' http://127.0.0.1:5000/login

On s'authentifie d'abord en utilisant la méthode POST et le endpoint /login. Cette requête nous renvoie le résultat suivant :

{"access\_token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2MjQ1MzEyOTcsIm5iZiI6MTYyNDUzMTI5NywianRpIjoiY2FmZTQ5YTQtMmM0ZC00YzNhLThlNTQtNzVkMjI0MTQ3MGYyIiwiZXhwIjoxNjI0NTMzMDk3LCJpZGVudGl0eSI6ImRhbmllbGRhdGFzY2llbnRlc3QiLCJmcmVzaCI6ZmFsc2UsInR5cGUiOiJhY2Nlc3MifQ.FCr57NYfbvlfwNXyqgleMzhz1GeXyds93AxO-WKPE1I"}

Vous devriez avoir un autre résultat si vous utilisez une clé secrète différente. En copiant l'access\_token obtenu, nous pouvons avoir accès aux deux routes protégées.

Exécutez cette commande

curl -X GET -H 'Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2MjQ1MzEyOTcsIm5iZiI6MTYyNDUzMTI5NywianRpIjoiY2FmZTQ5YTQtMmM0ZC00YzNhLThlNTQtNzVkMjI0MTQ3MGYyIiwiZXhwIjoxNjI0NTMzMDk3LCJpZGVudGl0eSI6ImRhbmllbGRhdGFzY2llbnRlc3QiLCJmcmVzaCI6ZmFsc2UsInR5cGUiOiJhY2Nlc3MifQ.FCr57NYfbvlfwNXyqgleMzhz1GeXyds93AxO-WKPE1I' http://127.0.0.1:5000/user

En utilisant l'en-tête Authorization : Bearer suivi du jeton, nous avons accès aux différentes ressources. Le résultat est le suivant :

{"logged\_in\_as":"danieldatascientest"}

On fait de même pour accéder à la ressource de l'autre route avec cette commande :

curl -X GET -H 'Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2MjQ1MzEyOTcsIm5iZiI6MTYyNDUzMTI5NywianRpIjoiY2FmZTQ5YTQtMmM0ZC00YzNhLThlNTQtNzVkMjI0MTQ3MGYyIiwiZXhwIjoxNjI0NTMzMDk3LCJpZGVudGl0eSI6ImRhbmllbGRhdGFzY2llbnRlc3QiLCJmcmVzaCI6ZmFsc2UsInR5cGUiOiJhY2Nlc3MifQ.FCr57NYfbvlfwNXyqgleMzhz1GeXyds93AxO-WKPE1I' http://127.0.0.1:5000/resource

Ce qui nous donne le résultat suivant :

{"owner":"danieldatascientest","resource":"Module DE"}

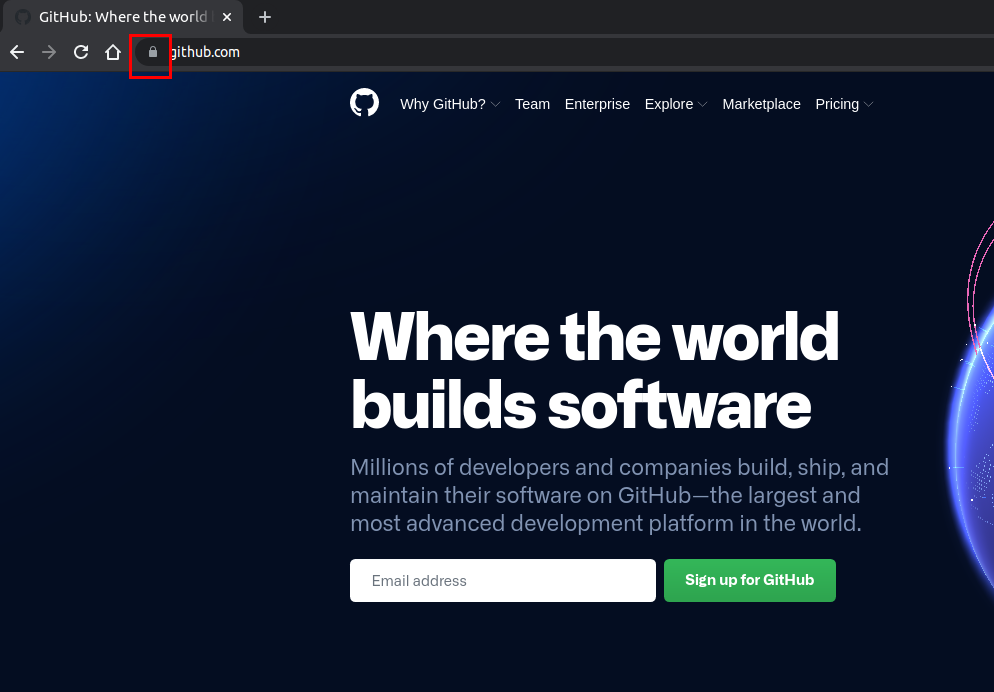
Nous avons vu la mise en place des JWT avec Flask, nous allons attaquer dans le prochain cours la méthode HTTPS.

## V. HTTPS

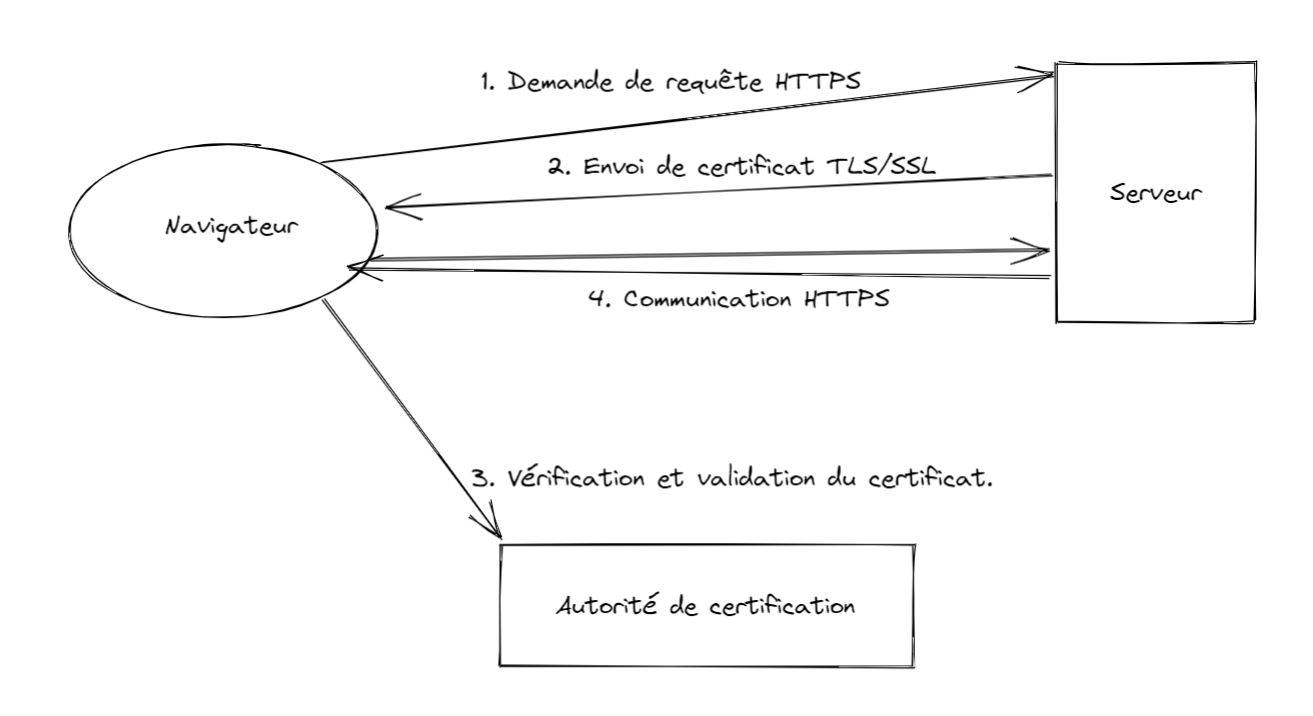
HTTPS signifie **Hypertext Transfer Protocol Secure**. C'est une extension du protocole HTTP dans laquelle les données sont cryptées et échangées via une connexion sécurisée. En utilisant une connexion sécurisée telle que Transport Layer Security (TLS) ou Secure Sockets Layer (SSL), la confidentialité et l'intégrité des données sont préservées et l'authentification des sites Web est également validée.

HTTPS assure la sécurité des données sur le réseau - principalement les réseaux publics comme Internet. HTTP n'étant pas crypté, ce protocole est vulnérable aux attaquants qui peuvent intercepter les données échangées entre le serveur et le client. En vertu de cela, le cryptage HTTPS est effectué de manière bidirectionnelle, ce qui signifie que les données sont cryptées à la fois côté client et côté serveur. Seul le client peut décoder les informations qui proviennent du serveur. Ainsi, HTTPS chiffre les données entre un client et un serveur, ce qui protège contre l'écoute clandestine, la falsification d'informations et la falsification des données.

Lorsque vous naviguez sur internet, vous pouvez vous assurez que les informations transitent bien via le protocole HTTPS en vérifiant la barre d'adresse qui porte le nom du site, grâce à l'icône de verrouillage généralement situé dans le coin supérieur gauche (la localisation de l'icône peut différer en fonction des navigateur) :



En termes simples, les données échangées entre l'utilisateur et le site ne sont pas lues, volées ou falsifiées par un tiers.



Comme l'illustre le schéma ci-dessus, pour utiliser HTTPS, nous avons besoin de certificat SSL/TLS qui va permettre une connexion sécurisée en la chiffrant. Une fois la demande de requête, le navigateur va vérifier la validité du certificat auprès de ce que l'on appelle une autorité de certification (qui est également celle qui fournit les certificats). Une fois le certificat validé, le navigateur peut communiquer avec le serveur en toute sécurité.

L’utilisation de HTTPS donc est indispensable afin de protéger et de chiffrer les données qui sont échangées. Nous aborderons dans ce cours les moyens pour mettre en place HTTPS. Pour plus d'informations sur HTTPS, vous pouvez lire [cette bande dessinée](https://howhttps.works/).

VI. Conclusion

Nous avons vu les principaux concepts et méthodes à connaitre afin de sécuriser des APIs.

En premier lieu, la sécurisation repose sur 3 piliers :

* L'authentification
* L'autorisation
* La traçabilité

Différentes méthodes peuvent être utilisées pour garantir la sécurité. Nous avons notamment vu les méthodes HTTP Basic Auth, les clés API, l'utilisation de JSON Web Token, etc.

Il en existe d'autres, comme les standards OAuth et OpenID Connect, et le choix d'une méthode se fera en fonction des spécifications et du type d'API.

Il faut également noter qu'on utilisera en général plusieurs méthodes conjointement. Par exemple, même si l'on utilise une méthode d'authentification, il est dans tous les cas recommandé d'échanger les données entre le client et le serveur via le protocole HTTPS.