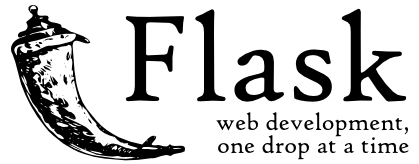
Flask

## **3. Prise en main de Flask**



### **a. Présentation de Flask**

[Flask](https://flask.palletsprojects.com/en/2.0.x/) est un célèbre framework Python pour réaliser des applications web. Sortie en 2011, Flask a su s'imposer comme une référence pour créer une API sur Python. Elle se démarque de [Django](https://www.djangoproject.com/), un autre célèbre framework utilisé en Python, par sa légèreté. En effet, Flask est considéré comme un **micro framework**, vous n'avez pas à définir les bases de données à utiliser ou la mise en forme. Cependant avec Flask, vous créez le noyau de votre application web, ensuite selon votre besoin vous y ajoutez différents services. C'est pourquoi, en quelques lignes, nous pouvons afficher facilement un Hello World sur une page web sans passer par un fichier HTML.

De plus, Flask étant **open-source**, vous êtes certains d'avoir une communauté qui vous aidera pour les erreurs que vous rencontrez mais qui continue à améliorer Flask. Les statistiques sur leur [dépôt Github](https://github.com/pallets/flask/tree/main/examples/tutorial) parlent d'elles-mêmes.

Ainsi, pour toutes les raisons citées, il s'agit du parfait outil pour créer sa première API.

### **b. Installation de Flask et première API**

Pour installer Flask, il suffit de passer par pip.

pip install Flask

Copiez les lignes suivantes dans un fichier first\_app.py :

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello World !"

Ensuite, définissez la variable d'environnement FLASK\_APP en donnant le nom de votre fichier python :

export FLASK\_APP=first\_app.py

Lancez la commande suivante :

flask run --host=0.0.0.0

Maintenant, vous devrez avoir le résultat sur votre console :

\* Serving Flask app 'first\_app' (lazy loading)

\* Environment: production

WARNING: This is a development server. Do not use it in a production deployment.

Use a production WSGI server instead.

\* Debug mode: off

\* Running on http://192.168.1.44:5000/ (Press CTRL+C to quit)

Sur un navigateur web allez à l'adresse adresse\_ip\_vm:5000 (localhost:5000 si vous utilisez votre machine personnelle), vous devrez apercevoir un Hello World !

 Vous n'avez pas à définir la variable d'environnement FLASK\_APP si vous appelez votre fichier python app.py ou wsgy.py, vous pouvez lancer directement flask run

 Le port 5000 est utilisé par défaut sur Flask, vous pouvez changer le port via le flag --port lorsque vous utilisez flask run

Observons en détail, ce que nous avons fait ici. Tout d'abord, nous avons instancié un objet de la classe Flask. Puis, avec cet objet nommé app, nous l'utilisons dans le décorateur @app.route("/"), pour créer une route (*endpoint*). En allant, sur /, nous avons le résultat de la fonction hello, qui retourne simplement un *Hello World !*. Ainsi, nous pouvons créer plusieurs routes en utilisant la variable app

Définissez la route "/daniel" qui affiche *"Hello Daniel !"*

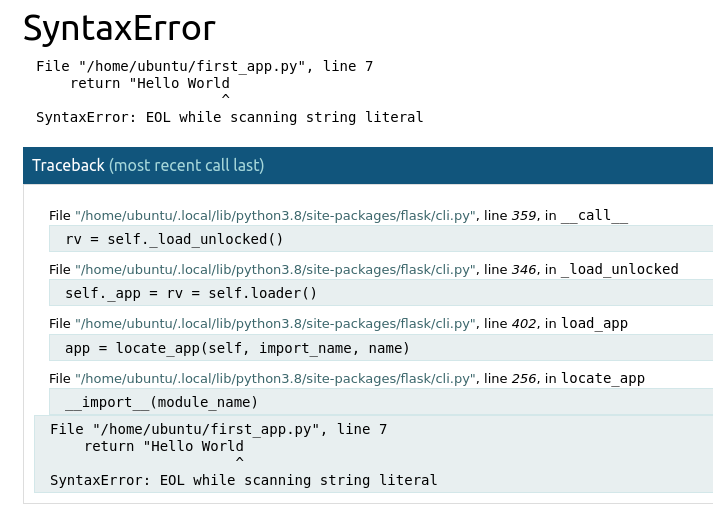
@app.route("/daniel")

def hello\_daniel():

return "Hello Daniel !"

### **c. Le mode DEBUG et le format des données envoyées**

Il est très vite ennuyeux d'arrêter à chaque fois le processus et de relancer l'API à chaque modification du fichier python, c'est pourquoi il existe le mode DEBUG. Une fois, que vous avez modifié et sauvegardé votre script python, en rafraîchissant la page web, vous obtenez le changement voulu. Pour activer ce mode, il faut mettre la variable d'environnement FLASK\_ENV à development. Par ailleurs, une fois que ce mode est activé, si votre API rencontre une erreur, l'exception Python lancée sera affichée sur la page Web. Ci-dessous, un guillemet était absent dans la chaîne de caractère renvoyée par la fonction et Flask nous l'informe.



Définissez la variable d'environnement FLASK\_ENV, lancez l'instruction flask run et modifiez le script par exemple, au lieu de Hello World, mettez Hello Votre-Prénom.

export FLASK\_ENV=development

flask run --host=0.0.0.0

Il y a une autre façon de lancer une application Flask. Elle n'est plus mise en avant dans la documentation car elle est considérée comme obsolète, toutefois elle reste encore très utilisée par habitude et vous la retrouverez sur de nombreux projets Python. De plus, de cette manière, nous n'avons pas besoin de définir les variables d'environnement FLASK\_APP et FLASK\_ENV.

À la fin de notre fichier first\_app.py, rajoutez les lignes suivantes :

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

app.run(host="0.0.0.0",debug=True)

Maintenant exécutez de la manière suivante :

python3 first\_app.py

Vous pouvez maintenant retrouver votre application web comme précédemment. Après il est recommandé de passer par flask run pour le déploiement de votre application mais aussi parce que le mode DEBUG y est optimisé.

Pour éviter de définir à nouveau les variables FLASK\_APP et FLASK\_ENV, vous pouvez aussi utiliser la bibliothèque python-dotenv installable avec pip. Il suffira simplement d'ajouter un fichier .env dans le dossier de votre projet Flask et dans ce fichier, il faudra définir les variables de la manière suivante :

FLASK\_APP=first\_app.py

FLASK\_ENV=development

Le format des fonctions n'est pas restreinte au format string. Si vous écrivez du code HTML, l'affichage y sera adapté. Ici, nous avons ajouté des balises pour écrire un titre en HTML, observez le résultat sur votre navigateur web.

@app.route("/html")

def hello\_html():

return "<h1>Hello World !</h1>"

De même, il est possible d'envoyer un dictionnaire, cela sera affiché comme un fichier JSON.

@app.route("/json")

def hello\_json():

return {"sequence": "Hello World"}

En passant par curl, nous pouvons voir cette différence avec le Content-Type, comme nous l'observons ci-dessous :

curl -X GET -i localhost:5000

HTTP/1.0 200 OK

Content-Type: text/html; charset=utf-8

Content-Length: 11

Server: Werkzeug/2.0.1 Python/3.8.10

Date: Tue, 28 Sep 2021 13:49:12 GMT

Hello World

curl -X GET -i localhost:5000/json

HTTP/1.0 200 OK

Content-Type: application/json

Content-Length: 32

Server: Werkzeug/2.0.1 Python/3.8.10

Date: Tue, 28 Sep 2021 13:51:07 GMT

{

"sequence": "Hello World"

}

curl -X GET -i localhost:5000/html

HTTP/1.0 200 OK

Content-Type: text/html; charset=utf-8

Content-Length: 20

Server: Werkzeug/2.0.1 Python/3.8.10

Date: Tue, 28 Sep 2021 13:52:02 GMT

<h1>Hello World</h1>

Attention, vous ne pouvez pas envoyer tout type de données, vous obtiendrez une erreur si vous envoyez un int par exemple. Les types acceptés sont les chaînes de caractères, les dictionnaires et les tuples de la forme (message,code) ou (message,code,en-tête).

### **d. Comment envoyer les différentes requêtes HTTP avec Flask ?**

Du cours précédent, nous avons vu les différentes méthodes liées aux requêtes HTTP, nous allons voir comment les appliquer sur Flask. Commençons par la méthode GET, cette méthode est la plus simple car c'est celle appliquée par défaut. Si vous observez le terminal où est lancé votre application Flask, à chaque fois que vous êtes allés sur les routes que vous avez définies, vous avez effectué une requête GET, comme vous l'atteste cet historique de requête.

Pour faire passer des données dans votre requête GET. La route suivante permet d'afficher Hello name avec name, le prénom inscrit dans l'adresse.

@app.route("/hello\_get/<name>")

def hello\_get(name):

return "Hello {}".format(name)

# Do not consider this line </name>

Contrairement à la méthode GET, les autres méthodes n'offrent pas la possibilité d'envoyer des informations depuis l'adresse fournie à votre navigateur web. Vous êtes contraints de passer par la commande curl.

Ajoutez la fonction ci-dessous à votre fichier python

from flask import request

@app.route("/hello\_post", methods=["POST"])

def hello\_post():

data = request.get\_json()

return "Hello {} \n".format(data["name"])

Envoyez le dictionnaire {"name":"Daniel"} au endpoint /hello\_post par curl et POST :

curl localhost:5000/hello\_post -d '{"name":"Daniel"}' -H 'Content-Type: application/json' -X POST

En comparant, avec la méthode GET, nous apercevons deux nouveautés. La première est l'ajout de l'argument methods, dans la création de notre route. Cet argument précise quelle méthode nous allons utiliser et par défaut il vaut GET.

La seconde différence rencontrée est l'utilisation de request. Par le biais de cet objet, nous récupérons les données envoyées par la méthode POST avec la fonction get\_json. Ensuite, il suffit simplement de manier le dictionnaire reçu dans le corps de la fonction.

Par ailleurs, il est commun qu'une même route soit accessible depuis plusieurs méthodes, d'où nous pouvons donner un tableau de méthodes et définir la route selon la méthode choisie. Pour cela, nous passons encore par request.

from flask import request

@app.route("/hello", methods=["POST","GET"])

def hello\_post\_get():

if request.method=="POST" :

data = request.get\_json()

return "Hello {} \n".format(data["name"])

return "Hello World ! "

Les méthodes PUT et DELETE, s'utilisent de la même manière que la méthode POST. Il faudra préciser la méthode dans la route avec methods et faire la requête avec curl.

 Dans la nouvelle version de Flask, vous pouvez préciser la méthode depuis le décorateur, à la place de @app.route vous pouvez avoir @app.post() pour la méthode POST. Toutefois, nous avons choisi de rester sur la définition la plus commune pour la suite du cours.

#### **Exercice d'application**

* PUT /add/users rajoutera l'utilisateur avec ses données dans la base de données.
* POST /update/users modifiera les données d'un utilisateur déjà existant, dans le cas contraire l'utilisateur sera ajouté dans la base de données.
* GET /users/<id> donnera les informations associées à l'utilisateur d'id.
* DELETE /delete/<id> supprimera les informations associées à l'utilisateur d'id

from flask import Flask

from flask import request

app = Flask(\_\_name\_\_)

user = {}

@app.route("/users/<id>")

def get\_user(id):

return user[id]

# Do not consider this line </id>

@app.route("/add",methods=["PUT"])

def add\_user():

data = request.get\_json()

user[str(len(user))]=data

return "L'utilisateur {} a été ajouté à la base de données.\n".format(len(user)-1)

@app.route("/update/<id>",methods=["POST"])

def update\_user(id):

if id in user:

user[id] = request.get\_json()

return "L'utilisateur {} a été modifié \n".format(id)

return "L'utilisateur {} est absent de la base de données \n".format(id)

# Do not consider this line </id>

@app.route("/delete/<id>",methods=["DELETE"])

def delete\_user(id):

if id in user:

del user[id]

return "L'utilisateur {} a été supprimé \n".format(id)

return "L'utilisateur {} est absent de la base de données \n".format(id)

# Do not consider this line </id>

## **4. Passer les données à une API**

Nous avons vu comment créer des points d'accès de notre API Flask avec les nombreuses méthodes HTTP. Il suffisait simplement d'utiliser le décorateur route, puis de définir ce que l'API faisait une fois arrivé au point d'accès.

### **a. Route dynamique**

Nous avions vu cette première manière d'envoyer les données :

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/hello/<name>")

def hello(name):

return "Hello {}".format(name)

# Do not consider this line </name>

Cependant, elle n'a pas été nommé, c'est ce que nous appelons, une route dynamique. En effet, contrairement à notre première route où nous affichions toujours Hello World, ici nous affichons Hello name avec name le prénom que nous indiquons depuis l'URI.

De plus, nous pouvons préciser le type des données que nous voulons.

@app.route("/age/<int:age>")

def hello(age):

return "Hello, I am {} years old".format(age)

# Do not consider this line </int:age>

En ne mettant pas un entier naturel, vous obtenez l'erreur 404. En effet, l'API attend un entier à la suite de /age/, donc si vous testez /age/Hello ou /age/0.3, l'API considérera que c'est une nouvelle route, sauf qu'elle n'existe pas. Nous verrons dans le prochain cours, comment envoyer un message approprié suite à une erreur.

Les types pour caractériser une donnée sont les suivantes :



### **b. Chaîne de requête**

Comme vous le savez, la méthode GET possède une spécificité pour envoyer les données, il s'agit d'une **chaîne de requête** (query parameters). En effet, à la fin de l'URI et à l'aide d'un ?, vous pouvez caractériser un ensemble de **clef-valeur**.

Pour exécuter ce type de requête, il faut utiliser Flask-Pydantic, téléchargeable depuis pip :

pip install Flask-Pydantic

Une fois cela fait, copiez les lignes ci-dessous, dans votre fichier python et lancez l'API.

from flask import Flask

from pydantic import BaseModel

from flask\_pydantic import validate

class Query(BaseModel):

name:str

age:int

app = Flask(\_\_name\_\_)

@app.route("/intro")

@validate()

def intro(query:Query):

return "Hello, my name is {} and I am {} years old".format(query.name,query.age)

Allez à l'adresse localhost:5000/intro?name=Daniel&age=33

Vous devrez apercevoir la phrase "Hello, my name is Daniel and I am 33 years old.". Comme à la précédente section, il s'agit d'une page qui est modifiée en accord avec les données transmises dans l'URI.

Pour arriver à ce résultat, nous pouvons repérer les nouveautés. La première est la création d'une classe héritant de BaseModel. Nous l'avons appelée ici Query, mais vous pouvez la nommer comme vous le souhaitez. Les attributs de la classe correspondent à ce que nous voulons envoyer par chaîne de requête. Flask fait une vérification de type, en effet si vous aviez mis dans age une chaîne de caractère, vous auriez eu une erreur de code 400.

Après avoir défini votre classe, il faut rajouter le décorateur @validate à la suite du décorateur de la route. Sans ce décorateur, vous obtiendrez une erreur Python.

Enfin la fonction que vous avez liée avec les décorateurs doit prendre en argument un objet de la classe héritée par BaseModel, appelé query (un nom différent de query pose problème). De cette variable query, vous récupérez les paramètres de la requête.

Créez une route affichant, sous le format JSON, que Damian Lillard est un meneur de jeu de Portland, né en 1990 à l'aide d'une chaîne de requête

from flask import Flask

from pydantic import BaseModel

from flask\_pydantic import validate

class Player(BaseModel):

name:str

position:str

team:str

birth\_year:int

app = Flask(\_\_name\_\_)

@app.route("/info")

@validate()

def info(query:Player):

return query.dict()

Interrogez cette route via curl :

curl -X GET -i "localhost:5000/info?name=Damian%20Lillard&position=PG&team=Por&birth\_year=1990"

### **c. Corps de requête**

L'idée est la même que précédemment, faire une nouvelle classe héritant de BaseModel, rajouter le décorateur @validate et récupérer les informations depuis l'objet. La seule différence est que nous n'utilisons pas la méthode GET et que nous devons nommer la variable body au lieu de query.

Il faudra configurer la classe en accord avec ce que vous voulez envoyer via la méthode POST et préciser dans le décorateur relatif à la route, qu'on envoie une méthode POST.

Réalisez l'exemple précédent en passant par un corps de requête :

from flask import Flask

from pydantic import BaseModel

from flask\_pydantic import validate

class Player(BaseModel):

name:str

position:str

team:str

birth\_year:int

app = Flask(\_\_name\_\_)

@app.route("/info",methods=["POST"])

@validate()

def info(body:Player):

return body.dict()

Maintenant, pour utiliser ce point de terminaison, lancez l'instruction suivante:

curl -X 'POST' -i \

'http://127.0.0.1:5000/info' \

-H 'Content-Type: application/json' \

-d '{

"name":"Damian Lillard",

"position":"PG",

"team":"Portland",

"birth\_year":1990

}'

Cette méthode est nettement plus sûre puisque nous n'avons pas des informations dans l'URI.

### **d. En-tête**

Des données sont présentes sur l'en-tête de réponse. Il arrive que vous avez besoin de récupérer ces informations. Pour cela, vous devez passer par l'objet request qui possède de nombreux attributs relatifs à cet en-tête, tel que User-Agent accessible par l'attribut user\_agent. La liste des attributs est disponible [ici](https://tedboy.github.io/flask/generated/generated/flask.Request.html#attributes).

Si vous ne voulez pas passer par les attributs de request, vous pouvez simplement utiliser l'attribut headers de request, qui vous renseignera le User-Agent de la manière suivante request.headers["User-Agent"].

Même si l'attribut headers se comporte comme un dictionnaire, son type ne l'est pas, il est de werkzeug.datastructures.EnvironHeaders. Vous pouvez le constater en créant une route qui affiche le type de request.headers et en utilisant curl.

Affichez l'en-tête en entier sur votre navigateur.

from flask import request

@app.route("/head")

def head():

return dict(request.headers)

Vous pouvez aussi envoyer votre propre en-tête et configurer la route selon ce que vous envoyez. Imaginons que nous avons la route suivante :

from flask import request

@app.route("/head")

def head():

if request.headers.get("username"):

return "Hello {}".format(request.headers.get("username"))

elif request.headers.get("enterprise"):

return "Hello {}".format(request.headers.get("enterprise"))

else:

return "Hello World !"

Avec l'instruction curl -i localhost:5000/head -H 'enterprise: DataScientest', vous aurez un Hello DataScientest, tandis qu'avec curl -i localhost:5000/head -H 'username: Daniel', vous obtiendrez un Hello Daniel.

## **5. Gestion des erreurs et des réponses**

Nous avons vu que le mode DEBUG nous affichait l'erreur rencontrée dans le script Python, mais cela n'aide pas l'utilisateur lambda. Il faut adapter le message d'erreur pour que l'utilisateur utilise correctement votre API. Par exemple, imaginons que vous avez implémenté une API d'un modèle de Machine Learning. Une variable de votre modèle attend une valeur comprise entre 0 et 10, toutefois l'utilisateur rentre une valeur égale à 1000, votre modèle de Machine Learning ne va pas fonctionner. Il faut alors envoyer une réponse adapté à l'utilisateur pour que l'utilisateur mette une valeur comprise entre 0 et 10.

De même, nous avions rencontré un code d'erreur 404 lorsque nous avions écrit dans une methode GET qui attendait un entier naturel au lieu d'une chaîne de caractère. Cependant, il fallait envoyer un message d'erreur BadRequest de code 400.

C'est pourquoi, lorsque vous créez une API, il faut faire aussi de la gestion d'erreur

### **a. Créer ses erreurs**

Il y a deux manières qui permettent d'agir lors de l'apparition d'une erreur. Le premier moyen est d'utiliser le décorateur @app.errorhandler(). Vous le mettez en amont de votre fonction comme avec le décorateur @app.route(). Il suffira de préciser le code de l'erreur ou le nom de la classe de l'erreur importé de werkzeug.exceptions en argument du décorateur errorhandler.

from flask import Flask

from werkzeug.exceptions import NotFound

app = Flask(\_\_name\_\_)

@app.route("/hello")

def hello():

return "Hello World"

@app.errorhandler(NotFound)

def handler\_error404(err):

return "You have encountered an error of 404",404

Maintenant, dirigez-vous sur une adresse renvoyant une erreur 404, par exemple l'adresse localhost:5000/helloname. Vous aurez le message de la fonction associée au décorateur errorhandler et dans l'historique des requêtes du terminal, vous aurez bien une erreur 404 (si vous ne précisez pas ce code, la requête aura le code 200). Vous pouvez réaliser une meilleure page pour indiquer l'erreur.

La variable err est issue de la classe NotFound, de ce fait vous pouvez faire une réponse plus développée par le biais de cette variable.

Le second moyen est de passer par la méthode register\_error\_handler() qui prend deux arguments. Le premier argument prend le code d'erreur ou le nom de la classe tandis que le second argument prend la fonction qui devra être appelée lors de l'erreur. La méthode doit être appelée sur la variable app.

Enlevez le décorateur errorhandler et utilisez à la place la fonction register\_error\_handler()

app.register\_error\_handler(404,handler\_error404)

À présent, nous allons observer comment envoyer des erreurs. Reprenons la route :

@app.route("/age/<int:val>")

def age(val):

return "Hello, I am {} years old".format(val)

# Do not consider this line </int:val>

Lorsque nous écrivons une chaîne de caractère à la suite de /age/, nous recevons une erreur 404 au lieu d'une erreur Bad Request représentant que l'utilisateur s'est trompé sur la syntaxe. Il y a plusieurs moyens de retourner cette erreur, la première est d'envoyer un tuple composé d'un message d'erreur et le code associé à notre erreur HTTP. Ici, il s'agit de 400. Avec la route ci-dessous, vous pouvez :

@app.route("/age/<val>")

def age(val):

try:

return "Hello, I am {} years old".format(int(val))

except ValueError:

return "You did not enter an integer",400

# Do not consider this line </val>

Nous pouvons aussi importer des exceptions correspondants aux erreurs HTTP de werkzeug.exceptions. Ici, nous devons récupérer l'exception BadRequest et la lancer à l'aide de raise.

from werkzeug.exceptions import BadRequest

@app.route("/age/<val>")

def age(val):

try:

return "Hello, I am {} years old".format(int(val))

except ValueError:

raise BadRequest("You did not enter an integer")

# Do not consider this line </val>

Enfin, nous pouvons simplement utiliser la fonction abort de flask et renseigner le message d'erreur via l'argument description de abort.

from flask import abort

@app.route("/age/<val>")

def hello(val):

try:

return "Hello, I am {} years old".format(int(val))

except ValueError:

abort(400,description="You did not enter an integer")

# Do not consider this line </val>

Créez une route "/random" qui retournera un nombre au hasard compris entre 0 et un entier end, mais l'accès est restreint à l'utilisateur qui enverra par POST "name":"Daniel", mais aussi la valeur de end. Il faudra donc envoyer l'exception HTTP associée à une mauvaise authentification mais aussi l'exception pour une valeur incorrecte pour end.

from flask import Flask,request

from werkzeug.exceptions import Unauthorized,BadRequest

import numpy.random as ran

app=Flask(\_\_name\_\_)

@app.route("/random",methods=["POST"])

def random():

data = request.get\_json()

if data["name"]=="Daniel":

try:

return "Your lucky number is {}\n".format(ran.randint(0,int(data["end"])+1))

except ValueError:

raise BadRequest("You did not enter an integer")

else:

raise Unauthorized("The user is not Daniel")

### **b. Créer ses réponses**

Nous avons vu que les formats acceptés d'une fonction associée à une route sont les dictionnaires, les chaînes de caractère et les tuples, mais en fait Flask convertit ses types sous un format spécifique à Flask qui est la classe Response. Pour utiliser cette classe, il est courant de passer par la fonction make\_response.

from flask import make\_response

@app.route("/test")

def test():

return make\_response("I use the function make\_response.")

Pour l'instant, rien n'est différent avec nos précédentes routes mais il est possible d'utiliser cette fonction pour associer un code d'état. Par défaut, le code est 200 mais de cette manière il est possible d'envoyer un message d'erreur. En effet, si nous avions écrit make\_response("You did not enter an integer",400), nous aurions reçu le message donné en argument avec l'erreur code 400 spécifique.

L'avantage principal de cette classe, c'est que vous pouvez inclure un en-tête personnalisé à votre réponse et donc envoyer des données de cette manière là. Avec la route ci-dessous, nous ajoutons à mon en-tête toutes les données envoyées par la méthode POST.

@app.route("/header",methods=["POST"])

def hd():

header = dict(request.headers)

data = request.get\_json()

for key,value in data.items():

header[key]=value

return make\_response("<H1>It's all good</H1>",200,data)

En lançant l'instruction ci-dessous, vous devrez apercevoir dans l'en-tête une nouvelle ligne avec name: Daniel :

curl -i localhost:5000/header -d '{"name":"Daniel"}' -H 'Content-Type: application/json' -X POST

Créez une route birth qui affichera la phrase "name was born in birth\_year". Il faudra renseigner les données name et birth\_year via la méthode POST. Si birth\_year n'est pas un nombre, il faudra renvoyer un message d'erreur (avec le bon code d'erreur associé) via make\_response avec un en-tête personnalisé.

@app.route("/birth",methods=["POST"])

def intro():

data = request.get\_json()

header = dict()

try:

int(data["birth\_year"])

return "{} was born in {}".format(data["name"],data["birth\_year"])

except ValueError:

header["status\_code"]=400

for key,value in data.items():

header[key]=value

return make\_response("{} is not an number".format(header["birth\_year"]),400,header)

## **6. Déploiement de modèle de Machine Learning**

Dans ce chapitre, nous allons prendre connaissance le déploiement de différents modèles de Machine Learning. En effet, vous allez lancer 3 APIs qui contiendra 3 différents modèles entrainés.

Une fois l'API lancée, on pourra faire appel aux modèles en requêtant l'API avec des données qui seront passées au modèles pour la prédiction.

On a trois différents types de modèle: Un modèle simple entraînée sur des données structurées, un modèle de NLP et un modèle de classification d'images

### **a) Un modèle de Machine Learning sur un jeu de données structuré**

Notre modèle a été entrainé sur [la dataset Titanic](https://www.kaggle.com/competitions/titanic/data) de Kaggle qui rassemble des données sur les passagers du Titanic, le but du modèle est de prédire à partir des caractéristiques d'un potentiel passager, s'il aurait survécu ou non au désastre du Titanic.

#### **Téléchargement du modèle**

Comme mentionné dans l'introduction, les modèles sont entraînés et sauvegardés en fichiers pickle, donc il faudrait les télécharger sur notre machine virtuelle.

Pour cela, Exécutez la commande suivante :

wget <https://dst-de.s3.eu-west-3.amazonaws.com/flask_fr/svm_titanic.pkl>

Pensez à mettre le modèle dans le même dossier que le script de l'API pour faciliter l'accès au fichier.

#### **Script de l'API**

Ici le script de l'API est fourni ci-dessous :

from flask import Flask, request, jsonify, json

from sklearn.svm import SVC

import pickle

import numpy as np

import pandas as pd

# Chargement du modèle

svm = pickle.load(open("svm\_titanic.pkl", "rb"))

app = Flask(\_\_name\_\_)

@app.route("/predict", methods=["GET"])

def predict():

features = []

data = request.get\_json(force=True)

features.append(data["pclass"])

features.append(data["sex"])

features.append(data["age"])

features.append(data["fare"])

features.append(data["embarked"])

features.append(data["single"])

features.append(data["famillySize"][0])

features.append(data["famillySize"][1])

features.append(data["famillySize"][2])

features.append(data["title"])

to\_predict = np.array([features]) # le vecteur de caractéristiques

prediction = svm.predict(to\_predict)[0]

if prediction==0:

result = {"Prediction" : "Survived"}

else:

result = {"Prediction" : "Did not survive"}

return jsonify(result)

Quelques explications de notre route /predict :

* Nous chargeons le modèle contenu dans le fichier svm\_titanic.pkl grâce à la méthode pickle.load
* Dans notre route, nous récupérons les valeurs de chaque variable du modèle depuis le corps de la requête (qui sera en format json). Nous construisons ensuite un vecteur pour le donner au modèle de Machine Learning, qui lui retournera une prédiction d'après les caractéristiques.

Installez la bibliothèque sklearn sur votre machine virtuelle ou machine locale afin de pouvoir utiliser le modèle.

pip install sklearn

Créez un fichier nommé svm\_api.py et copiez le code ci-dessus puis lancez l'API en mode développement sur le port 2222.

export FLASK\_APP=svm\_api.py

export FLASK\_ENV=development

flask run --host=0.0.0.0 --port=2222

#### **Utilisation de l'API**

Avant d'avoir entraîné notre modèle, nous avons passé par une partie de preprocessing et feature engineering ou nous avons sélectionné les variable du dataset et encodé celles-ci. On peut voir dans le script de l'api que nous avons ont tout 8 variables explicatives :

* **pclass** définit la classe du ticket. valeurs : [1, 2, 3].
* **sex** définit le sexe du passager: 0 pour male et 1 pour female.
* **age** définit la classe d'âge: 0 pour moins de 16 ans, 1 pour moins de 32 ans, 2 pour moins de 48 ans, 3 pour moins de 64 ans, 4 pour plus de 64 ans.
* **fare** définit la classe de prix: 0 pour moins de 7.91 dollars , 1 pour moins de 14.454, 2 pour moins de 31, 3 pour plus de 31.
* **embarked** définit le port d'embarcation: 0 = Southampton, 1 = Cherbourg, 2 = Queenstown.
* **single** définit si le passager a embarqué seul ou accompagné: 1 pour seul, 0 pour accompagné.
* **familySize** définit la taille de la famille du passager, la variable est en one-hot-encoding, donc la valeur de la clef doit être un vecteur: [1,0,0] pour une famille de 2, [0,1,0] pour une famille de 3 ou 4, [0,0,1] pour une famille de 5 ou plus
* **title** définit le titre du passager: {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Rare": 5}

Ouvrez une nouvelle console et interrogez l'API depuis curl en attribuant des valeurs concrètes aux attribut en format json dans le corps de votre requête.

curl -X GET -i http://localhost:2222/predict -d '{

"pclass": 1,

"sex": 0,

"age": 2,

"fare": 0,

"embarked": 1,

"single": 0,

"famillySize": [0,0,1],

"title": 1

}'

#### **Prédictions multiple à l'aide de fichier JSON**

Nous voulons faire plusieurs prédictions en même temps sans devoir envoyer une seule requête à chaque fois en rentrant les données manuellement.

Pour cela nous allons ajouter une nouvelle route au script de notre API, qui recevra un fichier json avec une liste de description des passagers et fera une prédiction pour chacun.

Ajoutez l'endpoint suivant dans le script de votre API svm\_api.py.

@app.route("/predictjson")

def predict\_json():

json\_data = request.files["json"] # récuperer le fichier envoyé par la requête

data = json.load(json\_data)

objects = data["data"] # charger les objets JSON

results = []

for obj in objects:

features = []

features.append(obj["pclass"])

features.append(obj["sex"])

features.append(obj["age"])

features.append(obj["fare"])

features.append(obj["embarked"])

features.append(obj["single"])

features.append(obj["famillySize"][0])

features.append(obj["famillySize"][1])

features.append(obj["famillySize"][2])

features.append(obj["title"])

to\_predict = np.array([features])

prediction = svm.predict(to\_predict)[0]

if prediction==0:

result = {"Prediction" : "Survived"}

else:

result = {"Prediction" : "Did not survive"}

results.append(result)

return str(results)

Créez un fichier test.json ou vous ajouterez au moins deux descriptions de passager.

Les objets doivent être contenus dans une liste "data" pour que l'API y accède. Lisez bien le script de l'endpoint.

{

"data": [

{

"pclass": 1,

"sex": 0,

"age": 2,

"fare": 0,

"embarked": 1,

"single": 0,

"famillySize": [0,0,1],

"title": 1

},

{

"pclass": 0,

"sex": 0,

"age": 1,

"fare": 0,

"embarked": 1,

"single": 0,

"famillySize": [0,1,0],

"title": 4

}

]

}

Interrogez l'endpoint predictjson de votre API en passant le fichier test.json dans votre requête avec la clef json

Pour envoyer un fichier avec votre requête GET, vous pouvez utiliser l'option -F de curl, voici un exemple curl -X GET -F "clef=@fichier.txt" http://localhost:8000/home

curl -F "json=@test.json" <http://localhost:2222/predictjson>

### **b) Modèle de Machine Learning sur données textuelles**

Dans cette partie, nous allons lancer une nouvelle API qui utilisera un modèle de Depp Learning de type **NLP**. Le modèle a été entraîné sur le jeu de données [Airline Tweets](https://www.kaggle.com/code/parthchittawar/airline-tweet-sentiment-prediction) pour faire une analyse de sentiments.

Comme pour la partie précédente, nous devons télécharger le modèle de Deep Learning mais aussi le tokenizer qui permettra à votre API d'effectuer des prédictions sur des données textuelles.

Dans un nouveau dossier tweets téléchargez les éléments suivant :

* Le tokenizer :

wget <https://dst-de.s3.eu-west-3.amazonaws.com/flask_fr/tokenizer.pkl>

* Le modèle :

wget <https://dst-de.s3.eu-west-3.amazonaws.com/flask_fr/model_tweets.h5>

Vu que le modèle est un réseau de neurones, il est nécessaire d'installer tensorflow sur votre machine. Nous le réalisons avec la commande pip install tensorflow

#### **Script python**

Dans le dossier tweets, créez un nouveau script python nlp\_tweets.py ou vous recopierez le programme suivant :

from flask import Flask, json, jsonify, request

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

import tensorflow as tf

import pickle

tokenizer = pickle.load(open("tokenizer.pkl","rb"))

model = tf.keras.models.load\_model("model\_tweets.h5")

max\_length = 90

app = Flask(\_\_name\_\_)

@app.route("/predict")

def predict():

data = request.get\_json(force=True)

tweet = data["tweet"] # extraction du tweet

sequence = tokenizer.texts\_to\_sequences([tweet])

to\_predict = pad\_sequences(sequence, max\_length, padding="post")

# Faire l'analyse de sentiment

prediction = model.predict(to\_predict)[0].argmax()

print(to\_predict)

if prediction==0:

result = {"sentiment": "negative"}

elif prediction==1:

result = {"sentiment": "neutral"}

else:

result = {"sentiment": "positive"}

return jsonify(result)

Les modules Tokenizer et pad\_sequences sont nécessaire pour le prétraitement des textes.

Lancez l'API sur le port 2222 de votre machine en mode développement.

export FLASK\_APP=nlp\_tweets.py

export FLASK\_ENV=development

flask run --host=0.0.0.0 --port=2222

#### **Utilisation de l'API**

Sur une nouvelle console, requêter l'API avec curl en passant la donnée textuelle sous format json avec l'attribut "tweet".

curl -X GET http://localhost:2222/predict -d '{

"tweet": "the trip was bad"

}'

#### **Prédictions multiples**

Comme pour la partie précédente, nous allons ajouter un endpoint qui va se charger de faire plusieurs prédictions depuis un fichier JSON

Ajoutez ce code au script de votre API:

@app.route("/predictjson")

def predict\_json():

json\_data = request.files["json"] # recuperer le fichier envoyé par la requête

data = json.load(json\_data)

tweets = data["data"]

results = []

for tweet in tweets:

sequence = tokenizer.texts\_to\_sequences([tweet])

to\_predict = pad\_sequences(sequence, max\_length, padding="post")

# Faire l'analyse de sentiment

prediction = model.predict(to\_predict)[0].argmax()

print(to\_predict)

if prediction==0:

result = {"sentiment": "negative"}

elif prediction==1:

result = {"sentiment": "neutral"}

else:

result = {"sentiment": "positive"}

results.append(result)

return str(results)

Dans le même dossier, créez un fichier test.json qui aura une liste "data" qui contiendra plusieurs chaînes de caractères qui représenteront les tweet à analyser.

{

"data": [

"the trip was good",

"the trip was bad"

]

}

Requêtez votre API en envoyant le fichier json avec curl.

curl -F "json=@test.json" <http://localhost:2222/predictjson>

### **c) Modèle de Machine Learning sur des images**

Cette troisième API aura pour but de mettre en production un modèle de Deep Learning pour la **Computer Vision**. Nous avons une architecture CNN qui détectera si dans l'image nous avons un père noël.

Le modèle à été entraîné sur une dataset d'image de pères noël et d'image qui représente des objets ou personnes aléatoires.

Créez un nouveau dossier santa où vous allez placer le modèle CNN que vous téléchargerez.

Téléchargement du modèle :

wget <https://dst-de.s3.eu-west-3.amazonaws.com/flask_fr/model_santa.h5>

Installez la librairie pillow avec la commande pip install pillow. Cette librairie est nécessaire pour le traitement des images que l'API recevra en requêtes.

#### **Script Python**

Une fois que la librairie est installée, vous pouvez construire l'API. Pour cela, créez un fichier santa\_api.py dans le dossier santa et insérez le script suivant:

from flask import Flask, json, jsonify, request

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.preprocessing import image

import numpy as np

import tensorflow as tf

from werkzeug.utils import secure\_filename

app = Flask(\_\_name\_\_)

model = tf.keras.models.load\_model("model\_santa.h5")

@app.route("/predict")

def predict():

# Sauvegarder l'image

image\_file = request.files['image']

filename = secure\_filename(image\_file.filename)

image\_file.save(filename)

# Charger l'image

test\_image = image.load\_img(filename, target\_size = (64, 64))

# Prétraitements

test\_image = image.img\_to\_array(test\_image)/255

test\_image = np.expand\_dims(test\_image, axis = 0)

# Faire la prédiction

proba = round(100\*model.predict(test\_image)[0][0], 2)

if proba < 50:

santa\_or\_not = 'Not Santa'

proba = round(100-proba, 2)

else:

santa\_or\_not = 'Santa'

return jsonify(

{

"prediction" : santa\_or\_not,

"probability" : proba

}

)

Ce script est chargé de :

* Importer les libraires nécessaire pour le bon fonctionnement de l'API
* Récupérer l'image envoyée dans la requête.
* Appliquer les prétraitements nécessaires à l'image que sont la normalisation des pixels et l'expansion de dimension.
* Faire appel au modèle CNN pour prédire la probabilité que l'image soit une image de père noël
* Retourner la prédiction sous forme de JSON

Lancez l'API sur le port 2222 de votre machine en mode développement.

export FLASK\_APP=santa\_api.py

export FLASK\_ENV=development

flask run --host=0.0.0.0 --port=2222

Utilisation de l'API

Sur une nouvelle console, interrogez l'API en passant une image avec la clef image dans la requête.

curl -F "image=@test.jpeg" <http://localhost:2222/predict>

7. Conclusion et évaluation

a. Conclusion

Nous avons vu comment avec Flask, nous pouvons réaliser rapidement une API en créant différentes routes. De plus, nous pouvons implémenter les différentes méthodes pour envoyer des requêtes HTTP en chaîne de requête, corps de requête et en-tête. Enfin, nous avons réalisé de la gestion d'erreur en renvoyant un message adapté mais aussi en envoyant des erreurs sur une mauvaise utilisation de l'API.

Par ailleurs, il est possible de réaliser des pages web dynamiques. Il suffit d'importer la fonction render\_template. Rajoutez un dossier templates dans votre projet et rajoutez-y le fichier HTML récupérable avec l'instruction suivante :

wget https://dst-de.s3.eu-west-3.amazonaws.com/flask\_fr/hello.html

Rajoutez la route suivante à votre API :

from flask import render\_template

@app.route("/hello\_render)

def html\_render():

return render\_template("hello.html")

La différence n'est pas frappante à lorsque nous envoyons simplement du code HTML. Toutefois il suffit de modifier le fichier HTML pour pouvoir utiliser des outils de programmation. Téléchargez le fichier ci-dessous :

wget https://dst-de.s3.eu-west-3.amazonaws.com/flask\_fr/table.html

Nous remarquons la présence de boucle for qui parcourt des listes. L'élément de la liste est accessible à l'aide de {{}}. Cette syntaxe particulière provient de Jinja qui est une bibliothèque qui fournit des template. La question est de savoir comment faire passer les listes dans notre fichier HTML, pour cela nous employons la fonction render\_template.

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

header = ["Name","Duty"]

data = [

["Daniel","Support"],

["Diane","Career Management"],

["Donna","Life of the program"]

]

@app.route("/")

def table():

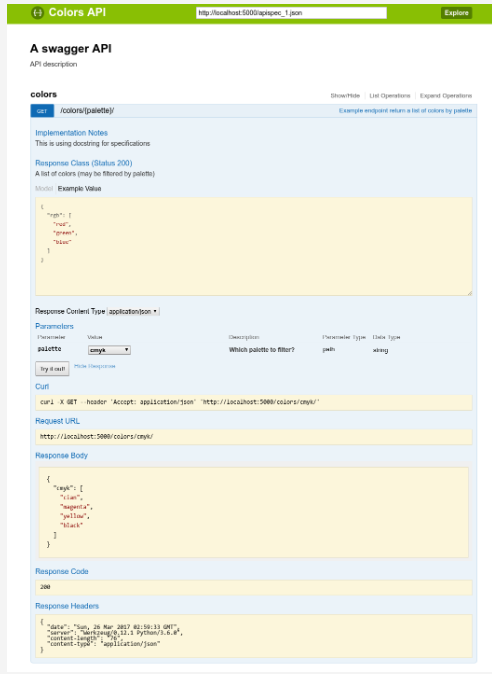
return render\_template("table.html",headings=header,data=data)

Dans la fonction render\_template, nous créeons les listes que nous avons définies dans le fichier HTML, ici il s'agit de headings et de data, puis nous donnons comme valeurs les listes que nous avons défini en amont. Vous pouvez faire passer de cette manière toutes les variables de votre fichier python.

Nous pouvons aussi créer un formulaire pour saisir nos données et les envoyer le tout avec Flask. Étant donné que Flask est un framework micro-service, vous pouvez trouver des extensions supplémentaires permettant de rajouter des options à votre besoin, ici pour les formulaires vous pouvez passer par WTForms.

Par exemple, vous pouvez coupler Flask avec l'extension Flasgger pour ajouter facilement une documentation à votre API. En effet, il est vital de fournir une documentation à votre API sinon votre application n'est pas utilisable par les personnes n'ayant pas contribué à la réalisation de l'API.

À l'aide de cette bibliothèque, vous obtenez une interface OpenAPI plus accessible comme ci-dessous :



### **b. Évaluation**

Pour l'évaluation, il faudra réaliser une API portant sur l'analyse de sentiment. Il suffira alors de renseigner la phrase dont nous voulons étudier le sentiment associé par des requêtes HTTP. L'API sera accessible via un système de pseudo/mot de passe fourni par le fichier credentials.csv récupérable à l'aide de la commande suivante :

wget https://dst-de.s3.eu-west-3.amazonaws.com/flask\_fr/credentials.csv

De ce fichier, il est visible que l'API possède deux modèles d'analyse de sentiment et que les utilisateurs n'ont pas tous accès aux mêmes versions.

De plus, le fichier credentials.csv indique aussi les modèles disponibles pour un utilisateur car l'API contiendra deux modèles d'analyse de sentiment. Imaginons qu'un utilisateur a v1=1 et v2=0, alors cela veut dire qu'il a accès au modèle v1 mais pas au modèle v2.

Tout d'abord, il faudra créer une route /status qui retounera 1 si l'API est fonctionnelle. Puis, ajoutez une route saluant l'utilisateur. Ensuite, faites une route /permissions indiquant quels modèles un utilisateur a accès.

La première version du modèle, disponible avec la route /v1/sentiment devra renvoyer un nombre aléatoire entre -1 et 1 tandis que la route /v2/sentiment devra renvoyer le compound d’un VaderSentiment.SentimentIntensityAnalyzer(). Pour plus d'informations sur ce modèle, vous pouvez vous diriger vers ce [lien](https://github.com/cjhutto/vaderSentiment#code-examples).

Voici un récapitulatif des routes à réaliser avec des spécificités dans les méthodes à utiliser.

* GET /status: renvoie 1 si l’API fonctionne.
* GET /welcome : renvoie un message de salutation avec le username indiqué par chaîne de requête.
* POST /permissions: renvoie la liste des permissions d’un utilisateur authentifié par son username et son password, rajoutez un en-tête à la réponse renseignant comportant le username, la valeur prise par v1 et v2.
* POST /v1/sentiment: renvoie le score de sentiment par le modèle v1 de la phrase proposée par l’argument sentence envoyé par POST si l’utilisateur s'est bien identifié par la paire (username,password). Cette paire devra être transmise par l'en-tête suivant {Authorization: username=password}, la paire devrait être encodée mais pour simplifier l'exercice, vous pouvez laisser tel quel.
* POST /v2/sentiment: Même fonctionnement que la précédente route, mais nous voudrons avoir le score retourné par le modèle v2

Une gestion des erreurs est attendue. Si un utilisateur s'est mal authentifié avec le code d'état adéquat, de même si l'utilisateur essaie d'utiliser un modèle dont il n'a pas les droits.

#### **Rendus**

Un fichier python app.py qui contient le programme de l’API développée avec Flask, un fichier requirements.txt qui représente l’environnement (Python) minimal qui permet de faire tourner cette API, ainsi qu'une documentation fournie sous forme de fichier texte ou de type pdf de votre API. Dans cette documentation, vous devrez fournir des exemples de requêtes pour pouvoir tester les différentes routes et aussi leurs résultats.