APIs avec FastAPI

1. Introduction aux APIs

a. Introduction

API signifie littéralement **Interface de Programmation d’Application** et est définie comme un ensemble de définitions de sous-programmes, de protocoles de communication et d'outils qui servent d’intermédiaire à la communication entre deux applications. Pour le Data Engineer et le Machine Learning Engineer, l’utilisation d’API va se révéler très utile lors des phases de déploiement et d'automatisation.

Une API est en réalité utilisée pour dé-corréler le service d'un client qui souhaite l'utiliser. Elle permet d'abstraire un fonctionnement complexe en donnant une liste de protocoles d'utilisation de ce service.

On peut penser à une API comme une prise de courant. Une prise de courant définit un protocole pour son utilisation: l'appareil que l'on souhaite brancher doit comporter deux broches, espacées d'un certain écart. Si cet écart est respecté alors la prise délivrera un courant de 220V à une fréquence de 50Hz. Du point de vue du client, c'est-à-dire de l'appareil électrique, la production de l'électricité n'a pas d'importance. Il peut s'agir de toutes sortes de centrales. De l'autre côté, c'est-à-dire du point de vue de la prise, on n'a pas besoin de s'adapter à l'appareil électrique: si le protocole est respecté, l'énergie nécessaire est fournie.

L'intérêt d'utiliser une API réside dans la simplification de l'utilisation d'un service complexe via une interface simple. Le terme d'API représente donc une interface qui abstrait, simplifie l'utilisation d'un service.

Il existe donc de nombreux types d'APIs possibles. Nous allons en évoquer quelques unes:

* la librairie [pandas](https://pandas.pydata.org/docs/reference/index.html#api) peut en fait être vue comme une API. On retrouve un ensemble de règles qui permettent d'utiliser des fonctions complexes à partir d'une documentation simplifiée.
* l'API [OpenWeatherMap](https://openweathermap.org/current) est une API de données. À partir d'une requête HTTP, on reçoit des données météorologiques dans le lieu demandé, à la date demandée.
* l'API [Google Cloud Translate](https://cloud.google.com/translate) permet d'utiliser la fonction de traduction de Google pour traduire facilement les textes que l'on utilise.

On peut voir que certaines APIs peuvent être très simples et simplement renvoyer certaines données alors que d'autres peuvent cacher l'utilisation de fonctions très complètes. On remarquera que dans le cas des APIs Google Translate et OpenWeatherMap, il faut s'identifier et éventuellement souscrire à un plan pour pouvoir accéder au service.

C'est en effet un des avantages très importants des APIs. En réécrivant clairement les règles d'utilisation d'une API, nous pouvons introduire l'utilisation de token d'authentification. On peut alors donner certains droits à certains utilisateurs, définir des limites dans l'utilisation de l'API et éventuellement vendre l'accès à l'API.

De plus, ces APIs s'inscrivent dans la philosophie de l'architecture micro-service, dans laquelle les composants d'un système doivent être les plus petits possibles et les plus isolés pour pouvoir être facilement mis à jour sans changer l'architecture globale du système.

Imaginons, par exemple, une entreprise qui propose des vélos à la location. Cette entreprise stocke ses données sur la disponibilité des vélos à la location dans une base de données. Cette base de données est interrogée régulièrement par différents outils: le site internet de l'entreprise, l'application mobile, qui permettent tous deux de réserver des données, par les bornes de location qui notifient du retour des vélos ainsi que par un dashboard interne. Si l'entreprise n'emploie pas une API, alors il faudra mettre à jour tous les outils lorsque l'entreprise sera amenée à changer sa base de données (par exemple pour permettre une utilisation distribuée de celle-ci). Avec une API, il suffit de mettre à jour l'API qui jouera le rôle d'intermédiaire. De plus, on pourra développer certaines fonctionnalités qui permettront à des acteurs extérieurs d'accéder à certaines données (on peut penser aux autorités locales qui pourraient utiliser ces données pour faire un suivi du service).

De la même façon, si on construit un algorithme d'analyse de sentiment performant, c'est-à-dire qui arrive à prédire correctement le sentiment positif ou négatif d'une phrase ou d'un texte, on pourra cacher le fonctionnement de cet algorithme par une API. Ainsi, nous pourrons construire des abonnements avec diverses limites d'utilisation de notre algorithme. De plus, le jour où l'algorithme est remplacé par un autre plus puissant, son utilisation ne change pas pour l'utilisateur final.

L'utilisation d'APIs présente donc de nombreux avantages: elles permettent de différencier les accès aux services en fonctions des utilisateurs, de simplifier les mises à jour du service dont l'API est l'interface.

Certaines entreprises reposent fortement sur ce principe d'APIs. Un exemple très célèbre est Amazon: en [2002](https://nordicapis.com/the-bezos-api-mandate-amazons-manifesto-for-externalization/), Jeff Bezos aurait demandé à toutes ces équipes de toujours créer des APIs à chaque fois qu'ils créent une fonctionnalité ou qu'ils exposent de la donnée.

On peut voir que les APIs sont donc très intéressantes d'un point de vue architectural mais aussi d'un point de vue plus métier car elles offrent des perspectives économiques.

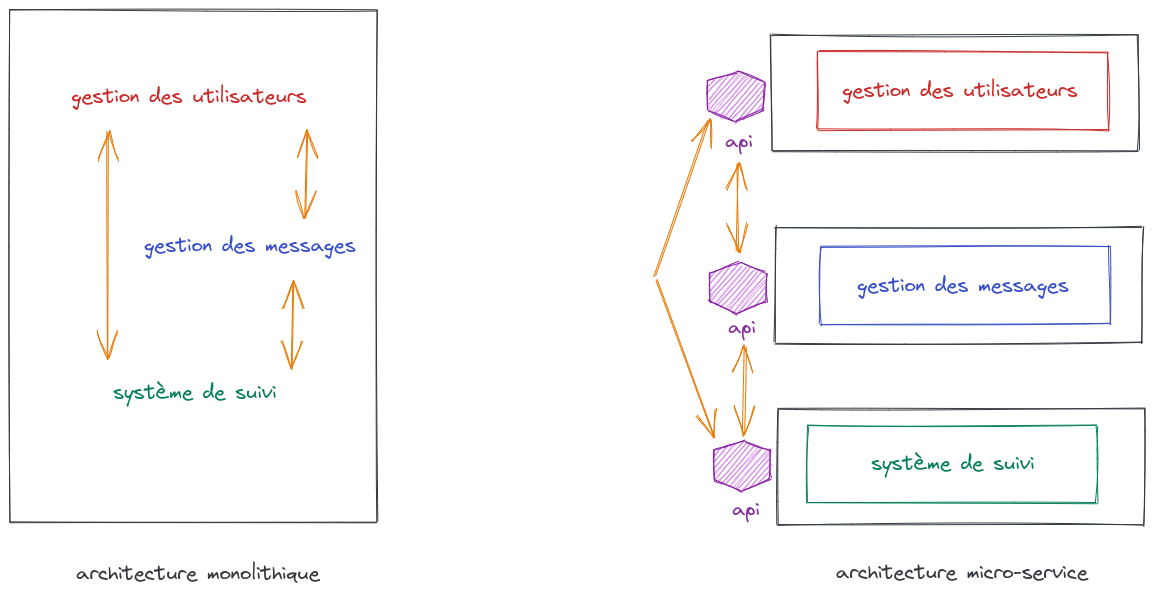
Enfin, il faut noter que pour garantir l'utilisation de ces APIs, les protocoles de communications doivent être très clairs et bien documentés sur les paramètres à fournir ainsi que sur leurs effets.

b. Architecture micro services

Avec des applications de plus en plus complexes, les organisations se tournent de plus en plus vers des architectures micro-services, c'est-à-dire des architectures dans lesquelles les services sont très découplés.

Prenons l'exemple d'un réseau social. L'entreprise a besoin d'un système d'identification et de gestion des utilisateurs, un système qui permet de créer et poster des messages, de systèmes de monitoring, etc... Une solution classique consisterait à créer une sorte de script très complexe qui permet de tout gérer d'un coup: de la réception des données jusqu'à leur stockage en passant par l'implémentation des droits. On parle alors d'architecture monolithique.

L'autre approche consisterait à créer une architecture de stockage. On va séparer la gestion des utilisateurs de la gestion des posts, etc... Ces différents services pourront communiquer entre eux grâce à des APIs.



L'architecture micro-service présente de nombreux avantages:

* les différents services sont **autonomes**, peuvent être développés par différentes équipes, dans différents langages puisqu'elles peuvent communiquer entre elles via leur API.
* chaque service est **spécialisé** dans la résolution d'une tâche unique ce qui permet de plus facilement identifier des problèmes ou de faire évoluer un service
* comme chaque service est une petite unité indépendante du reste du système et spécialisée, on peut facilement modifier ce service donnant ainsi des cycles de développement plus rapides et plus agiles.
* si certains services requièrent des ressources (stockage, calcul) plus importantes, ou si leur besoin en ressources évolue au fil du temps, on peut facilement faire **monter en puissance** un service sans déranger le fonctionnement des autres services.
* un service peut facilement être **réutilisé** par d'autres équipes dans d'autres projets. L'API permet en effet d'avoir une documentation précise de l'utilisation du service qui permet à d'autres équipes d'inclure le service dans leurs applications.
* on peut enfin facilement identifier d'où viennent les pannes dans un tel système et réparer ces pannes rapidement ou même de modifier le service pour que la panne ne réapparaisse plus.

## 2. Rappels sur les requêtes HTTP

### a. Éléments d'une requête HTTP

Le protocole HTTP est le protocole utilisé pour le Web. Créé en 1990, il permet de demander à un serveur de renvoyer certaines données ou de prendre certaines actions. Sa version sécurisée HTTPS n'est qu'une version chiffrée de HTTP.

Une requête HTTP est constituée de différents éléments.

En premier lieu, on retrouve l'URI (Universal Resource Identifier). Il s'agit de l'adresse à laquelle doit être effectuée la requête. Elle est généralement constituée du protocole utilisé, d'un nom de domaine ainsi que d'un point de terminaison (endpoint).

Par exemple, l'URI suivante http://example.org/resource peut se lire comme suit:

* http:// est le protocole utilisé
* example.org est le nom de domaine du serveur, c'est-à-dire une simplification de l'adresse IP du serveur.
* /resource est le point de terminaison que l'on souhaite solliciter

On pourra noter qu'une URI peut aussi comporter des paramètres. Par exemple, l'URI http://example.org/resource?key1=value1&key2=value2 permet de faire passer les clefs key1 et key2 qui ont respectivement les valeurs value1 et value2 au serveur. On parle alors de query string ou query parameters, traduit dans ce cours par chaîne de requête ou paramètres de requête.

On doit alors aussi préciser une méthode pour requêter les données. Les méthodes les plus utilisées sont GET, POST, PUT, DELETE, ...

Une requête peut aussi contenir un corps. (En général les requêtes de type GET ne peuvent pas comporter de corps). Le corps d'une requête permet de faire passer des données à la requête.

Enfin une requête peut aussi comporter des en-têtes (headers). Ces en-têtes contiennent les métadonnées relatives à la requête: type des données dans le corps de la requête, cookies, token d'authentification, ...

### b. Réponses HTTP

Lorsqu'une requête HTTP est émise, le serveur renvoie une réponse au client. Cette réponse est elle aussi composée de différents éléments:

* des en-têtes avec les métadonnées de la réponse
* un corps avec le contenu de la réponse
* un code d'état

Le contenu d'une réponse HTTP est très souvent du HTML pour des sites internet mais on retrouvera plus facilement du JSON ou du XML pour les APIs.

Le code d'état permet de comprendre facilement si la requête a réussi. Par convention, les codes d'erreur doivent correspondrent aux états suivants:

* 100: Information
* 200: Signifie que la requête a conclu / Succès
* 300: Redirection
* 40X: Signifie une erreur côté client / Erreur client
* 50X: signifie une erreur côté serveur / Erreur serveur

Ainsi, un code 404 est une erreur du client qui n'a pas rentré la bonne adresse pour accéder à la ressource alors qu'une erreur 503 sera une erreur du serveur qui n'arrive pas à faire tourner le service demandé.

### c. Liens avec le web

Les sites internet fonctionnent sur ce même principe d'architecture serveur-client que l'on requête via HTTP en utilisant un navigateur Web. Ainsi, en cliquant sur [ce lien](https://datascientest.com/), le navigateur envoie une requête de type GET auprès du serveur du site DataScientest. Si l'adresse est bonne, le serveur renvoie une réponse qui contient un fichier HTML qui sera interprété ensuite par le navigateur.

Lorsque l'on remplit un formulaire sur un site internet, il s'agit généralement d'une requête de type POST. La requête contient alors les données remplies dans le formulaire.

### d. Protocole HTTPS

Le protocole 'HTTPS' est une version plus sécurisée que le protocole HTTP. C’est en fait le protocole HTTP auquel est ajoutée une couche de chiffrement SSL (Secure Socket Layer). Il protège l’authentification d’un serveur, la confidentialité et l’intégrité des données échangées, et parfois l’authentification du client: une clef publique est donnée au client pour que les données renvoyées vers le serveur soient chiffrées; ces données sont alors décodées grâce à une clef privée disponible sur le serveur. Il tend à s’imposer comme la norme, poussée par les moteurs de recherche qui référencent mieux les sites utilisant un protocole HTTPS.

### e. Clients HTTP

Comme vu dans les exemples précédents, le navigateur est un client qui permet d'effectuer des requêtes HTTP vers des serveurs qui sont capables de renvoyer des données selon la requête. Toutefois, il existe d'autres outils plus simples à utiliser lorsque l'on souhaite interagir avec une API.

#### **CURL**

Nous allons interroger une API depuis le terminal à l’aide de l’interface en ligne de commande cURL (client URL Request Library). Pour faire une requête HTTP depuis le terminal avec cURL, la syntaxe pour faire une requête est la suivante :

curl -X GET http://example.com

L’argument -X introduit la méthode, ici, GET. Ensuite, nous pouvons écrire l’URI. L’API que nous allons interroger est une API web factice pour développeurs. Vous pouvez consulter ici la documentation de cette API.

curl -X GET <https://jsonplaceholder.typicode.com/posts/>1

Avec cette requête, nous recevons un objet JSON "stringifié" contenant des informations sur un message. Ici, nous avons demandé le post d’ID 1. Nous pouvons obtenir tous les posts avec une requête GET au point de terminaison (endpoint) /posts. C’est un cas de figure que nous rencontrons couramment: le endpoint avec un ID renvoie une observation individuelle tandis que sans aucun ID, il renvoie toutes les observations.

La réponse HTTP à cette requête contient seulement le corps. Pour voir les en-têtes, nous pouvons ajouter l’argument -i à la commande :

curl -X GET -i <https://jsonplaceholder.typicode.com/posts/1>

Dans l’en-tête, vous pouvez voir des informations sur le contenu de la réponse comme par exemple le code d’état qui vaut ici 200.

Nous avons vu comment faire une requête avec la méthode GET. Maintenant, si vous voulez utiliser la méthode PUT pour ajouter des données sur l’API, vous allez probablement vouloir préciser un corps et des en-têtes. Pour cela, il faut précéder vos en-têtes de l’argument -H et le corps de l’argument -d.

curl -X PUT -i\

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

-d '{"id": 1, "content": "hello world"}'\

<https://jsonplaceholder.typicode.com/posts/1>

Ici, nous indiquons dans l’en-tête le type de données envoyées (JSON), et dans le corps les données en question. cURL est sans doute l'outil le plus simple à utiliser et c'est celui que nous proposerons d'utiliser dans la suite du cours.

#### **Postman**

**Postman** est une plateforme collaborative pour le développement des API. Elle permet de construire et d'exécuter des requêtes HTTP, de les stocker dans un historique afin de pouvoir les rejouer, et de les organiser en collections.

Entre autres, Postman permet :

* d’envoyer rapidement et facilement des requêtes REST, SOAP et GraphQL
* d’automatiser les tests manuels et de les intégrer dans une pipeline CI / CD pour s’assurer qu'aucune modification de code n'interrompra l'API en production.
* de communiquer le comportement attendu d'une API en simulant les points de terminaison et leurs réponses sans avoir à configurer un serveur backend.
* de générer et publier une belle documentation lisible par machine pour rendre l’API plus facile à utiliser.
* de rester à jour sur la santé de son API en vérifiant les performances et les temps de réponse à intervalles planifiés.
* de collaborer en temps réel avec le contrôle de version intégré.

Postman propose donc une interface très simple pour faire des requêtes HTTP complexes. C'est un outil très utile pour tester/explorer une API. De plus, les fonctionnalités de Postman permettent d'exporter facilement une requête effectuée avec Postman dans le langage de votre choix.

Vous pouvez télécharger Postman [ici](https://www.postman.com/downloads/)

#### **Librairies HTTP de Python**

La librairie la plus simple pour effectuer des requêtes avec Python est sans doute la librairie [Requests](https://docs.python-requests.org/en/master/).

On pourra l'installer en utilisant le gestionnaire de package de Python

pip3 install requests

# or conda install requests

import requests

# creating a GET request

r = requests.get('https://jsonplaceholder.typicode.com/posts/1')

# getting the response elements

response\_dict = r.json

response\_header = r.headers

status\_code = r.status\_code

On peut bien sûr passer un corps, des en-têtes grâce à cette librairie.

r = requests.put(

url='https://jsonplaceholder.typicode.com/posts/1',

data={"id": 1, "content": "hello world"},

headers={"Content-Type": "application/json"}

)

Une autre librairie populaire est la librairie [Urllib](https://docs.python.org/3/library/urllib.html) installée nativement avec Python mais plus difficile à utiliser.

e. La norme REST

Chaque API possède une architecture spécifique, ainsi que des règles à respecter qui détermine les formats de données et commandes acceptées pour communiquer avec. Afin de favoriser l’accessibilité et la standardisation des API pour les développeurs, il existe désormais des architectures d’API classiques qui sont très souvent utilisées.

Par exemple, **l'architecture REST** (Representational State Transfer) est une architecture qui est très souvent utilisée dans la création de services WEB. Elle permet aux applications de communiquer entre elles quel que soit le système d’exploitation via le protocole HTTP. Une **API REST** utilise les requêtes HTTP pour communiquer et doit respecter les principes suivants :

* **Architecture client-serveur** : le client doit faire des requêtes HTTP pour demander des ressources. Il y a indépendance entre le côté client et l’application serveur de manière à ce que les modifications apportées à un point de terminaison n'affectent pas les autres.
* **Interface uniforme** : architecture simplifiée et qui permet à chaque partie d’évoluer indépendamment. Pour parler d’interface uniforme, il faut respecter les 4 contraintes :
* **Identification des ressources dans les requêtes** : les ressources sont identifiées dans les requêtes et sont séparées des représentations retournées au client.
* **Manipulation des ressources par des représentations** : les clients reçoivent des fichiers qui représentent les ressources. Ces représentations doivent contenir suffisamment d'informations pour être modifiées ou supprimées.
* **Messages auto descriptifs** : tous les messages renvoyés au client contiennent assez d'informations pour décrire la manière dont celui-ci doit traiter les informations.
* **Hypermédia comme moteur du changement des états applicatifs (HATEOAS)** : après avoir accédé à une ressource, le client REST doit être en mesure de découvrir toutes les autres actions disponibles par des hyperliens.
* **Avec mise en cache** : le client doit pouvoir mettre en cache les données que l’API fournie en réponse (https://aws.amazon.com/fr/caching/)
* **Système en couches** : la communication peut s’effectuer à travers des serveurs intermédiaires (serveurs proxy ou dispositifs de répartition de charge).
* **Sans état** : aucune information n’est stockée entre deux requêtes et l’API traite ainsi chaque requête comme une première demande

À noter que dans la norme REST, le endpoint doit désigner un objet que l'on souhaite manipuler et les méthodes doivent correspondre aux effets suivants:

* GET pour récupérer des informations
* POST pour ajouter des nouvelles données au serveur
* PUT pour modifier des données déjà présentes sur le serveur
* DELETE pour supprimer des données

Cette norme n'est pas la seule et n'est pas obligatoire à mettre en oeuvre pour obtenir une API efficace mais c'est sans doute la plus connue.

3. Introduction à FastAPI

a. La librairie FastAPI

La librairie [FastAPI](https://fastapi.tiangolo.com/) est une librairie très intéressante pour développer des APIs avec Python. En effet, les APIs sont relativement rapides par rapport à d'autres frameworks Python. De plus, FastAPI permet d'implémenter facilement une documentation ainsi que des contraintes de type sur les données.

b. Première implémentation

Dans cette partie, nous allons voir les principes de base de FastAPI. La première consiste à installer les librairies fastapi et uvicorn. uvicorn, qui est une librairie qui permet de lancer le serveur créé par FastAPI.

 Les librairies Python de création d'API utilisent généralement un serveur autre pour lancer l'API. Par exemple, on peut lancer une API Flask sans uvicorn mais ce n'est en général pas recommandé (cf le message de lancement).

Installez les librairies en utilisant la commande suivante

pip3 install fastapi uvicorn

Pour créer une API, nous allons devoir instancier la classe FastAPI du package fastapi.

Créez un fichier appelé main.py et collez les lignes suivantes

from fastapi import FastAPI

api = FastAPI()

@api.get('/')

def get\_index():

return {'data': 'hello world'}

Une fois que ce fichier est sauvegardé, lancez l'API dans une autre console en exécutant la commande suivante

uvicorn main:api --reload

Ici, on précise le fichier main et le nom de l'API à lancer à l'intérieur de ce fichier: api. L'argument --reload permet de mettre à jour automatiquement l'API lorsqu'on effectue des changements du fichier source. Dans la console, on doit observer la ligne suivante:

INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)

Cette ligne nous donne l'adresse à laquelle l'API fonctionne.

Dans une autre console, lancez la commande suivante pour faire une requête sur le endpoint /

curl -X GET <http://127.0.0.1:8000/>

Le résultat correspond bien à ce qu'on a passé comme valeur à return:

{ "data": "hello world" }

Exécutez la commande suivante pour afficher les en-têtes de la réponse:

curl -X GET -i <http://127.0.0.1:8000/>

On remarque que le contenu retourné est de type application/json: on n'a pas précisé cet argument mais FastAPI par défaut, renvoie des données au format json.

Sans arrêter l'API, modifiez le fichier main.py en remplaçant la fonction get\_index par les lignes suivantes:

def get\_index():

return "Hello world"

Relancez la commande curl précédente

On constate que le changement a bien été pris en compte par le serveur. Le type du contenu est toujours application/json.

Dans le code que nous avons exécuté on peut constater la présence d'un décorateur: @api.get('/'). Ce décorateur permet de préciser une route, c'est-à-dire un endpoint ainsi qu'une méthode. Ainsi, la fonction qui est décorée par cette ligne s'exécutera lorsqu'une requête de type GET est effectuée au endpoint /.

Cette façon de gérer les endpoints ainsi que les méthodes permet de voir facilement quelle fonction est appelée à quel moment et dans quelles conditions.

On peut bien sûr utiliser différentes méthodes et préciser différents endpoints.

Modifiez le fichier main.py en y mettant les routes suivantes

@api.get('/')

def get\_index():

return {

'method': 'get',

'endpoint': '/'

}

@api.get('/other')

def get\_other():

return {

'method': 'get',

'endpoint': '/other'

}

@api.post('/')

def post\_index():

return {

'method': 'post',

'endpoint': '/'

}

@api.delete('/')

def delete\_index():

return {

'method': 'delete',

'endpoint': '/'

}

@api.put('/')

def put\_index():

return {

'method': 'put',

'endpoint': '/'

}

@api.patch('/')

def patch\_index():

return {

'method': 'patch',

'endpoint': '/'

}

Exécutez les commandes suivantes pour tester toutes les routes

# GET at /

curl -X GET -i http://127.0.0.1:8000/

# POST at /

curl -X POST -i http://127.0.0.1:8000/

# PUT at /

curl -X PUT -i http://127.0.0.1:8000/

# DELETE at /

curl -X DELETE -i http://127.0.0.1:8000/

# PATCH at /

curl -X PATCH -i http://127.0.0.1:8000/

# GET at /other

curl -X GET -i <http://127.0.0.1:8000/other>

En quelques lignes, on a pu créer des routes pour différentes méthodes et différents endpoints. Tentons à présent d'interroger un endpoint qui n'existe pas:

Exécutez la commande suivante dans une console

curl -X GET -i <http://127.0.0.1:8000/no_where>

On obtient bien une erreur de type 404 Not Found avec un contenu au format json: {"detail": "Not found"}. FastAPI gère donc assez gracieusement ces erreurs de routage.

### c. Documentation

Un des enjeux importants des APIs est de fournir une documentation précise qui permet une utilisation simple de l'API. FastAPI présente l'avantage de générer automatiquement cette documentation.

En utilisant un tunnel entre le port 8000 de la machine distante et le port 8000 de la machine locale, on peut ouvrir l'API dans un navigateur web.

Créez ce tunnel et ouvrez un navigateur web à l'adresse http://localhost:8000/ (ou en changeant le port si vous avez décidé de faire suivre les données sur un autre port)

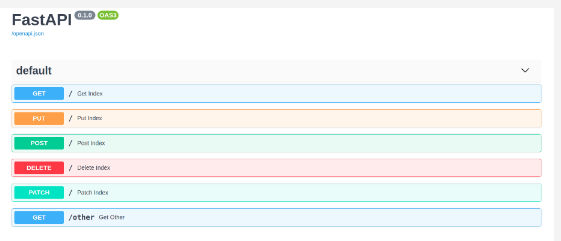
On devrait récupérer le résultat de la requête GET au endpoint /, c'est-à-dire, le résultat de la fonction get\_index:

{ "method": "get", "endpoint": "/" }

Ouvrez le endpoint docs dans le navigateur: <http://localhost:8000/docs>

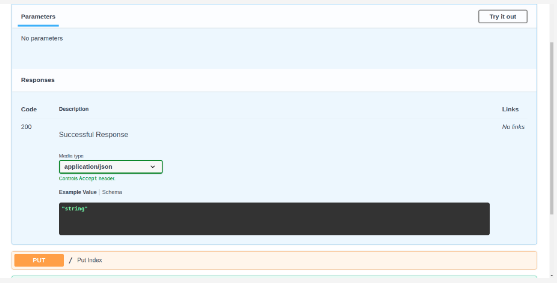
ssh -i data\_enginering\_machine.pem -L 8000:localhost:8000 [ubuntu@54.228.72.21](mailto:ubuntu@54.228.72.21)

Vous devriez arriver sur cette interface:



Il s'agit de l'interface OpenAPI (anciennement Swagger). Cette interface permet de voir facilement les endpoints et les méthodes acceptées.

Cliquez sur la méthode GET du endpoint /.



On peut cliquer sur le bouton Try it out puis execute pour lancer une requête GET sur le endpoint /. La réponse est retranscrite:



Cette interface nous donne la réponse, les en-têtes de la réponse. De plus, on peut voir la requête curl associée à l'essai qu'on vient de faire:

curl -X 'GET' \

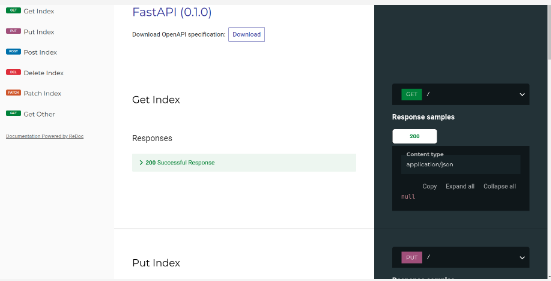
'http://127.0.0.1:8000/' \

-H 'accept: application/json'

On a donc un outil très utile pour faire des essais de notre API (à noter que les requêtes sont ici exprimées comme si elles étaient effectuées sur la machine hôte). FastAPI propose une autre version de cette interface au endpoint /redoc.

Ouvrez l'URL http://localhost:8000/redoc

Vous devriez arriver sur cette interface:



Cette interface est générée par [ReDoc](https://github.com/Redocly/redoc). Enfin, nous pouvons nous rendre au endpoint /openapi.json. On retrouvera la déclaration de l'API utilisée par ReDoc et OpenAPI pour générer les documentations:

Ouvrez l'URL <http://localhost:8000/openapi.json>

On devrait obtenir le json ci-dessous: ici, on l'a formatté pour le rendre plus lisible.

{

"openapi": "3.0.2",

"info": {

"title": "FastAPI",

"version": "0.1.0"

},

"paths": {

"/": {

"get": {

"summary": "Get Index",

"operationId": "get\_index\_\_get",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

},

"put": {

"summary": "Put Index",

"operationId": "put\_index\_\_put",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

},

"post": {

"summary": "Post Index",

"operationId": "post\_index\_\_post",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

},

"delete": {

"summary": "Delete Index",

"operationId": "delete\_index\_\_delete",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

},

"patch": {

"summary": "Patch Index",

"operationId": "patch\_index\_\_patch",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

}

},

"/other": {

"get": {

"summary": "Get Other",

"operationId": "get\_other\_other\_get",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

}

}

}

}

Dans la suite, nous verrons comment apporter plus d'informations à nos documentations.

## 4. Passer des arguments à une requête

FastAPI permet de gérer les arguments d'une requête plus facilement qu'avec Flask. Nous verrons dans cette partie comment utiliser le routage dynamique puis comment passer des arguments directement dans une requête. Enfin, nous verrons comment FastAPI génère la documentation relative aux arguments.

### a. Routage dynamique

Le routage dynamique permet de générer des endpoints de manière automatique.

Modifiez le fichier main.py en y collant les lignes suivantes

from fastapi import FastAPI

api = FastAPI(

title='My API'

)

@api.get('/')

def get\_index():

return {'data': 'hello world'}

Si l'API ne tourne plus, relancez-la en utilisant la commande

uvicorn main:api --reload

Nous allons créer un endpoint dynamique /item: on pourra alors ajouter un itemid à notre endpoint pour créer un nouvel endpoint:

Ajoutez les lignes suivantes à votre fichier main.py sans arrêter l'API

@api.get('/item/{itemid}')

def get\_item():

return {'route': 'dynamic'}

On peut à présent faire une requête sur ce endpoint dynamique:

Essayez ce nouveau endpoint avec différentes requêtes

curl -X GET -i http://127.0.0.1:8000/item/1

curl -X GET -i http://127.0.0.1:8000/item/my\_item

Nous allons à présent modifier notre fonction de manière à ce qu'elle prenne en compte la valeur passée dans le endpoint:

Remplacez la fonction get\_item par les lignes suivantes

@api.get('/item/{itemid}')

def get\_item(itemid):

return {

'route': 'dynamic',

'itemid': itemid

}

Réessayez les commandes précédentes.

On doit obtenir un résultat comme celui-ci:

{

"route": "dynamic",

"itemid": "my\_item"

}

On peut avoir des routes dynamiques complexes avec différents arguments. Par exemple:

@api.get('/item/{itemid}/description/{language}')

def get\_item\_language(itemid, language):

if language == 'fr':

return {

'itemid': itemid,

'description': 'un objet',

'language': 'fr'

}

else:

return {

'itemid': itemid,

'description': 'an object',

'language': 'en'

}

Ajoutez ces lignes au code source de l'API et essayez cette nouvelle route

FastAPI fournit de plus de nombreuses façons de contrôler le type des arguments fournis à l'API. Ainsi, si l'on souhaite que itemid soit forcément un nombre entier, on peut utiliser des annotations:

Remplacez la fonction get\_item par les lignes suivantes

@api.get('/item/{itemid:int}')

def get\_item(itemid):

return {

'route': 'dynamic',

'itemid': itemid

}

Essayez à présent les requêtes suivantes

curl -X GET -i http://127.0.0.1:8000/item/1234

curl -X GET -i http://127.0.0.1:8000/item/my\_item

On remarque que la deuxième requête renvoie une erreur 404: puisque my\_item ne correspond pas à un nombre entier, FastAPI ne connaît pas cette route. On peut avoir différentes routes qui existent pour un même endpoint dynamique et une même méthode si on veut prendre en compte les différents types de données.

Ajoutez les lignes suivantes au fichier source

@api.get('/item/{itemid:float}')

def get\_item\_float(itemid):

return {

'route': 'dynamic',

'itemid': itemid,

'source': 'float'

}

@api.get('/item/{itemid}')

def get\_item\_default(itemid):

return {

'route': 'dynamic',

'itemid': itemid,

'source': 'string'

}

Essayez les requêtes suivantes

curl -X GET -i http://127.0.0.1:8000/item/1234

curl -X GET -i http://127.0.0.1:8000/item/my\_item

curl -X GET -i http://127.0.0.1:8000/item/1.234

Notez que l'ordre dans lequel on définit les routes a une importance ! Si on définit le chemin par défaut, sans argument sur le type des données en premier, c'est cette fonction qui sera évaluée en premier et qui retournera le résultat.

Inversez l'ordre de définition des fonctions de manières à avoir get\_item\_default en premier et relancez les requêtes précédentes.

Pour s'entraîner à créer des routes dynamiques, nous allons essayer de créer une petite API qui renvoie des informations sur une base d'utilisateurs. Cette base sera représentée par une liste de dictionnaires, présentée ci-dessous.

users\_db = [

{

'user\_id': 1,

'name': 'Alice',

'subscription': 'free tier'

},

{

'user\_id': 2,

'name': 'Bob',

'subscription': 'premium tier'

},

{

'user\_id': 3,

'name': 'Clementine',

'subscription': 'free tier'

}

]

Les routes à créer sont les suivantes:

GET / renvoie un message de bienvenue

GET /users renvoie la base de donnée en entier

GET /users/userid renvoie toutes les données d'un utilisateur en fonction de son id. userid devra être un nombre entier. Si le userid fourni ne correspond pas à un utilisateur existant, on retournera un dictionnaire vide.

GET /users/userid/name renvoie le nom d'un utilisateur en fonction de son id. userid devra être un nombre entier. Si le userid fourni ne correspond pas à un utilisateur existant, on retournera un dictionnaire vide.

GET /users/userid/subscription renvoie le type d'abonnement d'un utilisateur en fonction de son id. userid devra être un nombre entier

L'API est à coder dans un fichier différent de main.py et à conserver puisque nous nous en servirons de nouveau dans la suite.

from fastapi import FastAPI

users\_db = [

{

'user\_id': 1,

'name': 'Alice',

'subscription': 'free tier'

},

{

'user\_id': 2,

'name': 'Bob',

'subscription': 'premium tier'

},

{

'user\_id': 3,

'name': 'Clementine',

'subscription': 'free tier'

}

]

api = FastAPI()

@api.get('/')

def get\_index():

return {

'greetings': 'welcome'

}

@api.get('/users')

def get\_users():

return users\_db

@api.get('/users/{userid:int}')

def get\_user(userid):

try:

user = list(filter(lambda x: x.get('user\_id') == userid, users\_db))[0]

return user

except IndexError:

return {}

@api.get('/users/{userid:int}/name')

def get\_user\_name(userid):

try:

user = list(filter(lambda x: x.get('user\_id') == userid, users\_db))[0]

return {'name': user['name']}

except IndexError:

return {}

@api.get('/users/{userid:int}/subscription')

def get\_user\_suscription(userid):

try:

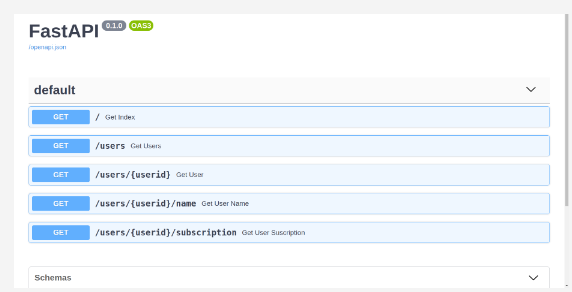
user = list(filter(lambda x: x.get('user\_id') == userid, users\_db))[0]

return {'subscription': user['subscription']}

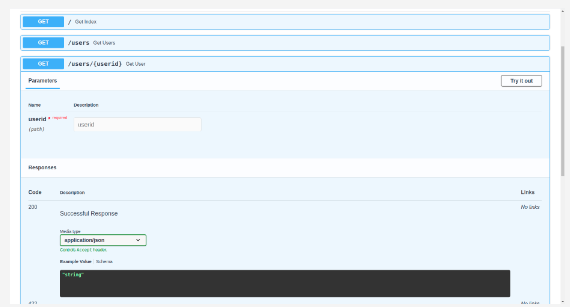
except IndexError:

return {}

En se rendant dans l'interface graphique de l'API, on obtient le menu suivant:



On remarque que l'interface pour utiliser ces requêtes permet de passer des arguments pour le routage dynamique:



### b. Chaîne de requête

Nous avons vu comment faire passer des données à l'API en utilisant le routage dynamique. Dans cette partie, nous allons voir comment le faire en utilisant la chaîne de requête . FastAPI permet en effet de préciser facilement quels arguments peuvent être passés via la chaîne de requête. Dans l'exemple suivant, nous allons définir une fonction qui pourra prendre un argument argument1. Cet argument devra être passé dans la chaîne de requête.

Remplacez le contenu du fichier main.py par les lignes suivantes

from fastapi import FastAPI

api = FastAPI()

@api.get('/')

def get\_index(argument1):

return {

'data': argument1

}

Exécutez la commande suivante pour appeler cette fonction

curl -X GET -i http://127.0.0.1:8000/?argument1=hello%20world

La fonction a bien accès aux données envoyées. Essayons à présent de faire la même requête sans spécifier l'argument argument1:

Exécutez la commande suivante

curl -X GET -i http://127.0.0.1:8000/

On obtient une erreur 422 Unprocessable Entity avec le contenu suivant:

{

"detail": [

{

"loc": ["query", "argument1"],

"msg": "field required",

"type": "value\_error.missing"

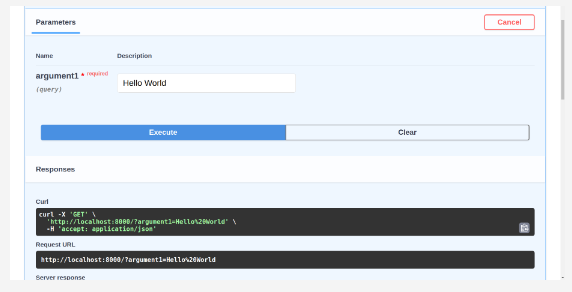
}

]

}

Cette réponse nous permet donc de comprendre pourquoi notre requête a échoué: le champ argument1 est manquant.

Ouvrez l'interface graphique OpenAPI et essayez la requête depuis cette interface



On voit dans l'interface que le champ argument1 est indispensable ("required"). En descendant dans l'interface, on peut aussi voir les codes que peuvent renvoyer cette route: 200 si la requête réussit et 422 si la requête ne présente pas le champ requis.

Comme avec le routage dynamique, on peut utiliser les annotations de Python pour contrôler le type des données envoyées.

Ajoutez les lignes suivantes au code source

@api.get('/typed')

def get\_typed(argument1: int):

return {

'data': argument1 + 1

}

Exécutez les requêtes suivantes

curl -X GET -i http://127.0.0.1:8000/typed?argument1=1234

curl -X GET -i http://127.0.0.1:8000/typed?argument1=hello

La deuxième requête génère une erreur 422 et renvoie le contenu suivant:

{

"detail": [

{

"loc": ["query", "argument1"],

"msg": "value is not a valid integer",

"type": "type\_error.integer"

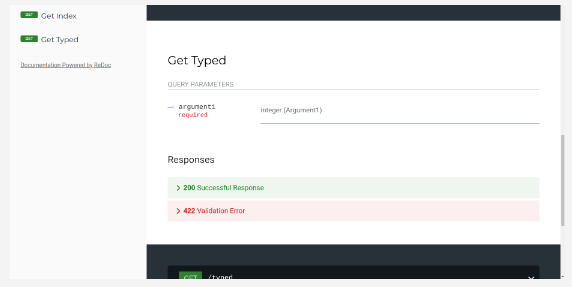
}

]

}

En utilisant l'interface graphique de OpenAPI, on peut voir qu'on ne nous laisse pas essayer un champ argument1 qui ne serait pas un nombre entier.

Ce typage des données est très précieux pour définir précisément l'utilisation d'une API: dans l'interface Redoc, on voit très vite quel type d'argument est utilisable pour quel endpoint:



On peut aussi voir sur le manifeste OpenAPI (disponible au endpoint /openapi.json) que le type des données est précisé (en fait l'interface redoc et OpenAPI est générée à partir de ce manifeste).

Enfin, nous pouvons choisir d'avoir un argument optionnel. Pour cela, nous pouvons utiliser la classe Optional de la librairie typing. On doit cependant proposer une valeur par défaut.

Ajoutez les lignes suivantes au fichier source

from typing import Optional

@api.get('/addition')

def get\_addition(a: int, b: Optional[int]=None):

if b:

result = a + b

else:

result = a + 1

return {

'addition\_result': result

}

En utilisant l'interface OpenAPI ou en utilisant curl, créez différentes requêtes pour essayer l'omission du paramètre b.

### c. Corps de la requête

Pour passer des données à l'API, la librairie FastAPI repose sur l'utilisation de la class BaseModel de pydantic pour expliciter la forme du corps de la requête.

Nous allons tout d'abord créer une classe Item héritée de la classe BaseModel.

Ajoutez les lignes suivantes à votre fichier source

from pydantic import BaseModel

from typing import Optional

class Item(BaseModel):

itemid: int

description: str

owner: Optional[str] = None

Ici la classe Item possède les attributs itemid qui doit être un nombre entier, description qui doit être une chaîne de caractères et owner qui est une chaîne de caractère optionnelle. Nous allons créer une route pour laquelle il faudra associer un corps de requête contenant ces attributs. Cette obligation se fera en utilisant les annotations dans la définition de la fonction:

Ajoutez ces lignes à votre fichier source pour définir une route prenant un Item comme corps

@api.post('/item')

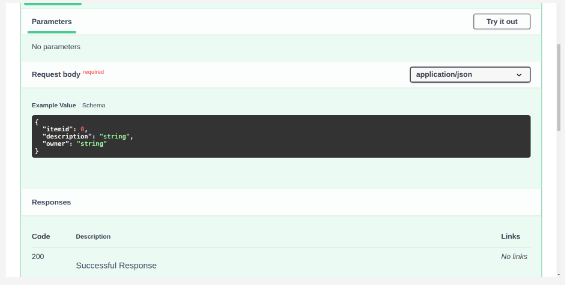
def post\_item(item: Item):

return {

'itemid': item.itemid

}

Ouvrez l'interface OpenAPI et inspecter la description de la route POST /item



On peut voir que les annotations données ici ne sont plus comptées comme des paramètres (des arguments à préciser dans la chaîne de requête). Par contre, on doit à présent préciser le corps de la requête ("Request body required"). Grâce à l'héritage de la classe BaseModel, on a pu changer l'interprétation de l'annotation par FastAPI.

On nous propose un exemple de données à passer directement généré par OpenAPI.

Exécutez les requêtes suivantes pour tester cette route en faisant bien attention aux codes d'erreur

curl -X 'POST' -i \

'http://127.0.0.1:8000/item'

curl -X 'POST' -i \

'http://127.0.0.1:8000/item' \

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

-d '{

"itemid": 1234,

"description": "my object",

"owner": "Daniel"

}'

curl -X 'POST' -i \

'http://127.0.0.1:8000/item' \

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

-d '{

"itemid": 1234,

"description": "my object"

}'

curl -X 'POST' -i \

'http://127.0.0.1:8000/item' \

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

-d '{

"itemid": 12345

}'

curl -X 'POST' -i \

'http://127.0.0.1:8000/item' \

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

-d '{

"itemid": 12345,

"description": "my object",

"other": "something else"

}'

La première requête renvoie une erreur 422 car elle ne comporte pas de corps. La deuxième requête comporte tous les champs et fonctionne donc correctement. La troisième possède tous les champs obligatoires. La quatrième renvoie une erreur 422 car elle ne possède pas le champ description. Enfin, la dernière fonctionne correctement. On remarque que les requêtes fonctionnent si on fournit bien les champs non optionnels de la classe Item.

Modifiez la fonction post\_item de manière à ce qu'elle renvoie l'attribut other de item et relancez la dernière requête

@api.post('/item')

def post\_item(item: Item):

return {

'itemid': item.other

}

Dans ce cas, on obtient une erreur 500 Internal Server Error: en effet, l'objet de la classe Item ne possède pas d'attribut other bien qu'il ait été passé dans le corps de la requête. Pour s'en convaincre, on pourra regarder la console dans laquelle l'API tourne: AttributeError: 'Item' object has no attribute 'other'.

Modifiez la fonction post\_item de façon à ce qu'elle renvoie l'objet item directement

@api.post('/item')

def post\_item(item: Item):

return item

Relancez la requête suivante

curl -X 'POST' -i \

'http://127.0.0.1:8000/item' \

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

-d '{

"itemid": 12345,

"description": "my object",

"other": "something else"

}'

L'utilisation de cette classe BaseModel en tant que classe mère permet donc à une route d'accepter un corps. On force le corps de la requête à respecter un certain schéma avec certaines valeurs qui peuvent être optionnelles. De plus, l'utilisation de cette classe permet d'ignorer des champs qui ne sont pas prédéfinis. Enfin, la classe BaseModel permet de renvoyer facilement tous les attributs du corps d'une requête qui ont été créés au format JSON sans avoir besoin de préciser cette définition.

Notez enfin que l'on peut utiliser les librairies typing et pydantic pour donner des types plus complexes à nos données. L'exemple suivant montre une utilisation de ces librairies.

from pydantic import BaseModel

from typing import Optional, List

class Owner(BaseModel):

name: str

address: str

class Item(BaseModel):

itemid: int

description: str

owner: Optional[Owner] = None

ratings: List[float]

available: bool

Pydantic permet aussi d'utiliser des types "exotiques" comme des URL http, des adresses IP, ... Si vous souhaitez explorer ces types de données, vous pouvez vous rendre à cette [adresse](https://pydantic-docs.helpmanual.io/usage/types/).

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

En reprenant l'API définie dans la partie a, nous allons faire quelques modifications pour mettre en application ce que nous venons de voir.

Il faut à présent ajouter les routes suivantes:

* PUT /users crée un nouvel utilisateur dans la base de données et renvoie les données de l'utilisateur créé. Les données sur le nouvel utilisateur doivent être fournies dans le corps de la requête.
* POST /users/userid modifie les données relatives à l'utilisateur identifié par userid et renvoie les données de l'utilisateur modifié. Les données sur l'utilisateur à modifier doivent être fournies dans le corps de la requête
* DELETE /users/userid supprime l'utilisateur désigné par userid et renvoie une confirmation de la suppression.

On choisira de renvoyer un dictionnaire vide dans le cas d'une erreur interne et on utilisera une classe User héritée de BaseModel.

# previous code is not included

from typing import Optional

from pydantic import BaseModel

class User(BaseModel):

userid: Optional[int]

name: str

subscription: str

@api.put('/users')

def put\_users(user: User):

new\_id = max(users\_db, key=lambda u: u.get('user\_id'))['user\_id']

new\_user = {

'user\_id': new\_id + 1,

'name': user.name,

'subscription': user.subscription

}

users\_db.append(new\_user)

return new\_user

@api.post('/users/{userid:int}')

def post\_users(user: User, userid):

try:

old\_user = list(

filter(lambda x: x.get('user\_id') == userid, users\_db)

)[0]

users\_db.remove(old\_user)

old\_user['name'] = user.name

old\_user['subscription'] = user.subscription

users\_db.append(old\_user)

return old\_user

except IndexError:

return {}

@api.delete('/users/{userid:int}')

def delete\_users(userid):

try:

old\_user = list(

filter(lambda x: x.get('user\_id') == userid, users\_db)

)[0]

users\_db.remove(old\_user)

return {

'userid': userid,

'deleted': True

}

except IndexError:

return {}

### d. En-têtes

Dans cette partie, nous allons voir comment faire passer des données au serveur via les en-têtes de la requête. Cette commande peut être très utile pour passer des tokens d'authentification ou vérifier le type de contenu, l'origine de la requête, ... Pour cela, nous allons pouvoir utiliser la classe Header de fastapi.

Par exemple, la fonction suivante permet de vérifier la valeur de User-Agent. Ce header est utilisé pour déterminer la source d'une requête:

from fastapi import Header

@api.get('/headers')

def get\_headers(user\_agent=Header(None)):

return {

'User-Agent': user\_agent

}

Collez ces lignes à la fin du fichier main.py, lancez l'API et essayez la requête suivante

curl -X GET -i http://127.0.0.1:8000/headers

La réponse devrait être:

{ "User-Agent": "curl/7.68.0" }

Utilisez l'interface OpenAPI pour essayer cette nouvelle route

On voit dans ce cas que le User-Agent retourné est le User-Agent du navigateur.

Conclusion

Nous avons vu dans cette partie comment faire passer des données depuis le client vers le serveur de 4 manières différentes:

* en utilisant le routage dynamique
* en utilisant les chaînes de requête
* en utilisant le corps de la requête
* en utilisant les en-têtes de la requête

## 5. Améliorer la documentation

Nous avons vu dans les parties précédentes que FastAPI permet de créer une vérification du type des données et de créer des règles quant aux données requises ou optionnelles pour faire requêter un endpoint. On peut ainsi générer facilement un manifeste (/openapi.json) ainsi qu'une documentation OpenAPI (/docs) ou ReDoc (/redoc).

Toutefois, il est intéressant d'organiser les connaissances ainsi que de documenter les arguments.

### a. Personnaliser la documentation des fonctions

Pour ajouter des commentaires à l'utilisation d'un endpoint, nous pouvons utiliser le docstring de la fonction. Nous pouvons aussi donner un nom à notre API via la classe FastAPI.

Remplacez le contenu du fichier main.py par les lignes suivantes

from fastapi import FastAPI

from fastapi import Header

api = FastAPI(

title="My API",

description="My own API powered by FastAPI.",

version="1.0.1")

@api.get('/')

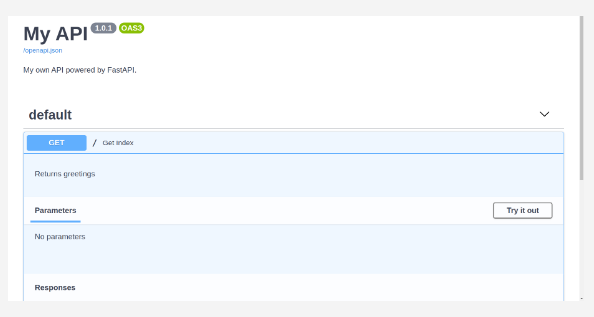
def get\_index():

"""Returns greetings

"""

return {'greetings': 'welcome'}

Sur l'interface OpenAPI, on peut voir les informations suivantes:



Le nom de l'API, sa version ainsi que sa description ont bien changé par rapport aux exemples précédents. De plus, on peut voir que la route GET / a maintenant une description issue tout droit du docstring de la fonction.

On peut aussi ajouter l'argument name au décorateur pour donner un nom à la route dans OpenAPI: par défaut, les noms des routes sont créés à partir des titres des fonctions utilisées: Get Index pour get\_index.

Remplacez le nom de la fonction get\_index dans le code source par Hello World

@api.get('/', name="Hello World")

def get\_index():

"""Returns greetings

"""

return {'greetings': 'welcome'}

### b. Documentation du corps des fonctions

Nous allons à présent voir comment FastAPI gère la définition de modèles issus de la classe BaseModel:

Ajoutez les lignes suivantes dans le fichier main.py et actualisez la page de la documentation OpenAPI

from pydantic import BaseModel

from typing import Optional

class Computer(BaseModel):

computerid: int

cpu: Optional[str]

gpu: Optional[str]

price: float

Rien n'a changé.

Ajoutez les lignes suivantes et actualisez la page

@api.put('/computer', name='Create a new computer')

def get\_computer(computer: Computer):

"""Creates a new computer within the database

"""

return computer

On voit que la deuxième route a été ajoutée. On remarque aussi que des Schémas sont apparus. On peut sélectionner le schéma de Computer pour comprendre comment est formée cette entité.

On remarque aussi que des erreurs sont disponibles: HTTPValidationError et ValidationError. Nous reviendrons sur ce point dans la suite. On peut ajouter une description à notre schéma en ajoutant un docstring à la classe Computer:

Remplacez la définition de la classe Computer par les lignes suivantes

class Computer(BaseModel):

"""a computer that is available in the store

"""

computerid: int

cpu: Optional[str]

gpu: Optional[str]

price: float

Dans l'onglet Schemas, la description de la classe Computer est maintenant disponible.

Les headers peuvent aussi avoir une description en utilisant l'argument description.

Ajoutez les lignes suivantes dans le fichier main.py et vérifiez la définition de la nouvelle fonction

@api.get('/custom', name='Get custom header')

def get\_content(custom\_header: Optional[str] = Header(None, description='My own personal header')):

return {

'Custom-Header': custom\_header

}

On peut voir dans la description de la fonction que l'en-tête est documenté.

### c. Organiser la documentation

Par défaut, l'ensemble des fonctions est documenté dans un onglet default. On peut cependant choisir d'organiser ces fonctions dans différentes parties. Pour cela, on doit spécifier l'argument tags dans le décorateur.

Remplacez le contenu du fichier main.py par les lignes suivantes

from fastapi import FastAPI

api = FastAPI(openapi\_tags=[

{

'name': 'home',

'description': 'default functions'

},

{

'name': 'items',

'description': 'functions that are used to deal with items'

}

])

@api.get('/', tags=['home'])

def get\_index():

"""returns greetings

"""

return {

'greetings': 'hello world'

}

@api.get('/items', tags=['home', 'items'])

def get\_items():

"""returns an item

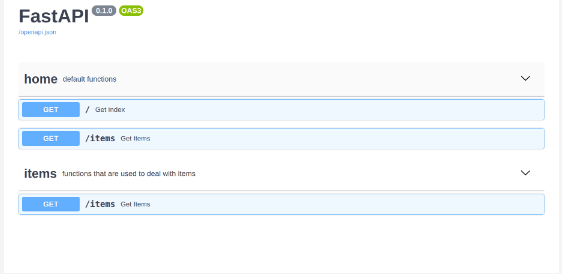
"""

return {

'item': "some item"

}

On obtient la documentation suivante:



Les fonctions sont maintenant réparties dans différentes parties. Une même fonction peut être mise dans plusieurs parties. On a pu ajouter une description pour les différentes parties en utilisant l'argument openapi\_tags du constructeur de la classe FastAPI.

On peut aussi changer l'adresse des documentations OpenAPI et Redoc en utilisant les arguments docs\_url ou redoc\_url. Si ces arguments sont passés à None, ces endpoints sont désactivés. On pourra enfin choisir de changer l'adresse du manifeste OpenAPI avec l'argument openapi\_url.

### d. Utilisation des erreurs

Les erreurs sont un outil important lorsqu'on développe des applications: elles permettent de donner facilement des indications sur une mauvaise manipulation de l'application. On va notamment pouvoir donner des informations en utilisant le code de statut de la réponse mais aussi en donnant des informations dans le corps de la réponse. Jusqu'à présent, nous avons pu obtenir des erreurs de types 500, 404 et 422. Nous allons voir dans cette partie comment créer des erreurs plus personnalisées.

Remplacez le contenu du fichier main.py par les lignes suivantes

from fastapi import FastAPI

api = FastAPI()

data = [1, 2, 3, 4, 5]

@api.get('/data')

def get\_data(index):

return {

'data': data[int(index)]

}

En ouvrant la documentation OpenAPI, on peut voir que deux erreurs sont proposées. Dans ce cas précis, nous pouvons voir que si la valeur d'index est supérieur à 4 ou inférieur à 0, on risque d'obtenir une erreur de type IndexError. De plus, si index n'est pas un nombre entier, on devrait là obtenir une ValueError.

Exécutez les deux commandes suivantes pour déclencher ces erreurs

curl -X GET -i http://127.0.0.1:8000/data?index=my\_index

curl -X GET -i http://127.0.0.1:8000/data?index=1234

Dans les deux cas, on obtient une erreur de type 500: Interval Server Error. L'erreur a lieu dans le code source de l'application mais FastAPI ne sait pas quelles informations sont renvoyées à l'utilisateur.

On a vu aussi que FastAPI générait lui-même ses erreurs pour des routes non trouvées (404: on pourra faire une requête GET /nowhere pour s'en convaincre) ou pour des formats de données qui ne correspondent pas aux attentes définies (422: notamment via l'utilisation d'annotations ou de classes héritées de BaseModel). Nous allons utiliser des blocs try-except pour récupérer les erreurs Python et renvoyer des HTTPException avec les codes HTTP associés.

Remplacez la définition de la fonction get\_data par les lignes ci-dessous

from fastapi import HTTPException

@api.get('/data')

def get\_data(index):

try:

return {

'data': data[int(index)]

}

except IndexError:

raise HTTPException(

status\_code=404,

detail='Unknown Index')

except ValueError:

raise HTTPException(

status\_code=400,

detail='Bad Type'

)

Relancez les requêtes pour vérifier les résultats

On peut donc facilement changer les codes d'erreurs et les données renvoyées lors d'une erreur. Pour l'argument detail, on peut donner un dictionnaire ou tout autre structure pouvant être interprétée comme un JSON.

Enfin, si l'on souhaite modifier la forme des données renvoyées lors de l'erreur, on peut créer nos propres exceptions et les faire passer dans le décorateur @api.exception\_handler.

Ajoutez ces lignes au fichier source

from fastapi import Request

from fastapi.responses import JSONResponse

import datetime

class MyException(Exception):

def \_\_init\_\_(self,

name : str,

date: str):

self.name = name

self.date = date

@api.exception\_handler(MyException)

def MyExceptionHandler(

request: Request,

exception: MyException

):

return JSONResponse(

status\_code=418,

content={

'url': str(request.url),

'name': exception.name,

'message': 'This error is my own',

'date': exception.date

}

)

@api.get('/my\_custom\_exception')

def get\_my\_custom\_exception():

raise MyException(

name='my error',

date=str(datetime.datetime.now())

)

Prenons un peu de temps pour décrire ce code. Dans le premier bloc, nous définissons une nouvelle Exception. On lui donne les attributs name et date. Dans le deuxième bloc, on indique à FastAPI comment réagir lorsque l'exception est soulevée. On donne une réponse de type JSON à renvoyer, en lui donnant un status\_code et un status. On peut ainsi accéder aux attributs de la requête ou de l'exception pour les renvoyer dans un JSON. Enfin le dernier bloc nous permet de définir une route qui génère cette erreur.

Essayez cette route pour voir le résultat

curl -X GET -i http://127.0.0.1:8000/my\_custom\_exception

Pour documenter les différentes erreurs possibles pour une route donnée, on pourra passer un dictionnaire contenant les codes d'erreur possibles ainsi que les descriptions associées.

Ajoutez les lignes suivantes au fichier main.py et consultez la documentation de la fonction Get Thing

responses = {

200: {"description": "OK"},

404: {"description": "Item not found"},

302: {"description": "The item was moved"},

403: {"description": "Not enough privileges"},

}

@api.get('/thing', responses=responses)

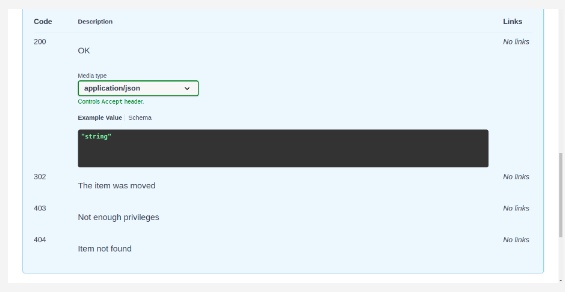
def get\_thing():

return {

'data': 'hello world'

}

On devrait obtenir la documentation suivante:



Ces descriptions ne changent pas le contenu de la fonction mais elles permettent de mieux comprendre les erreurs soulevées.

Exercice d'application

En reprenant l'API construite dans les parties précédentes, ajouter les codes d'erreur ainsi que des descriptions pour les différentes routes construites.

6. Performances

Dans les parties précédentes, nous avons vu les avantages que pouvait proposer FastAPI par rapport à une librairie comme Flask: on contrôle facilement les données qui sont transmises à l'API et on peut générer une documentation très facilement.

a. Performances simples

On peut aussi constater que FastAPI est un outil relativement rapide comparé à Flask. On est loin des frameworks dédiés au traitement des requêtes HTTP qui sont bien plus rapides mais on obtient des performances intéressantes en terme de rapidité des réponses. On pourra se référer aux benchmarks suivants:

* <https://web-frameworks-benchmark.netlify.app/result?l=python>
* <https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=json&l=zijzen-sf&a=2>

b. Concurrence

Un des avantages de FastAPI est de proposer une façon simple de traiter les requêtes de manière asynchrone. On peut ainsi lancer des traitements qui peuvent prendre du temps, sans pour autant bloquer l'accès à l'API. Pour définir une fonction asynchrone, il nous suffit ici d'utiliser la même syntaxe utilisée avec asyncio.

Dans cette partie, nous allons montrer l'avantage que peut présenter le fait de traiter les requêtes de manière asynchrone.

Dans un fichier fastapi\_async.py, collez les lignes suivantes:

from fastapi import FastAPI

import time

import asyncio

api = FastAPI()

def wait\_sync():

time.sleep(10)

return True

async def wait\_async():

await asyncio.sleep(10)

return True

@api.get('/sync')

def get\_sync():

wait\_sync()

return {

'message': 'synchronous'

}

@api.get('/async')

async def get\_async():

wait\_async()

return {

'message': 'asynchronous'

}

Lancez l'API nouvellement créée

Nous venons de créer une API qui présente deux points de terminaison /sync et /async. Ces deux points de terminaison ne font rien si ce n'est attendre pendant 10s puis renvoyer une réponse. On pourrait imaginer que la requête déclenche une opération longue sur une base de données. On veut tout de même recevoir une réponse rapide sur l'acceptation de cette commande.

Dans un autre fichier test\_requests.py, collez les lignes suivantes:

import requests

import time

from multiprocessing import Pool

def compute\_response\_time\_sync(x):

t0 = time.time()

requests.get(url='http://127.0.0.1:8000/sync')

t1 = time.time()

return t1 - t0

def compute\_response\_time\_async(x):

t0 = time.time()

requests.get(url='http://127.0.0.1:8000/async')

t1 = time.time()

return t1 - t0

def overflow\_requests(function, number\_of\_parallel\_operations=20):

with Pool(number\_of\_parallel\_operations) as p:

values = p.map(function, [i for i in range(

number\_of\_parallel\_operations)])

s = 0

for i in values:

s += i

return s/len(values)

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

print('making 20 requests on the `/sync` endpoint ...')

delta\_t = overflow\_requests(compute\_response\_time\_sync, 20)

print('took {} seconds'.format(delta\_t))

print('making 20 requests on the `/async` endpoint')

delta\_t = overflow\_requests(compute\_response\_time\_async, 20)

print('took {} seconds'.format(delta\_t))

Dans ce script, on interroge les deux endpoints. Pour simuler une arrivée simultanée, on utilise un Pool qui permet de paralléliser les requêtes. Ce script simulera donc l'interrogation de notre API par une vingtaine de clients.

Lancez le script avec la commande

python3 test\_requests.py

Sur une machine avec 16 coeurs, on constate que les premières requêtes prennent en moyenne 12 secondes. En effet, la machine doit traiter 20 requêtes mais n'a la possibilité d'en gérer que 16 à la fois. Il reste alors 4 requêtes qui prendront le double de temps, soit 20 secondes: 10 \* 16 + (10 + 10) \* (20 - 16).

En revanche, on voit que les requêtes asynchrone sont quasiment instantanées. Ici, on a choisi de ne pas attendre la réponse de la fonction asynchrone pour envoyer une réponse. Mais dans le cas où on choisit d'attendre la réponse, en ajoutant le terme await devant wait\_async, les requêtes prendront toutes 10s.

Modifiez la ligne wait\_async() par await wait\_async() puis relancer le test

Les requêtes prennent toutes 10s.

L'intégration des fonctions asynchrones se fait très facilement et rien n'empèche de systématiser l'utilisation du mot-clef async devant toutes les fonctions appelées aprés des décorateurs de méthode ou même des décorateurs de gestion d'erreur.

## 7. Conclusion et évaluation

### a. Conclusion

Nous avons vu dans ce cours que FastAPI est un outil très utile pour créer des APIs. Une des clefs d'une API performante est une documentation exhaustive, facile à prendre en main et FastAPI permet simplement de mettre en place une telle documentation. De plus, l'utilisation de OpenAPI (anciennement Swagger) comme documentation permet de facilement mettre en place des tests.

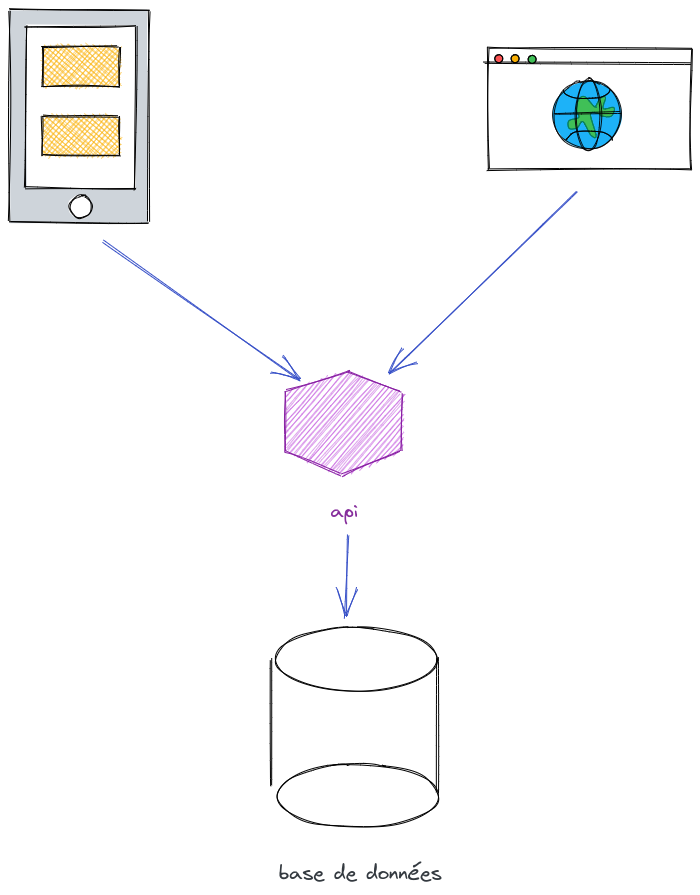
FastAPI semble aussi plus facile à prendre en main dans sa syntaxe que Flask et plus explicite sur sa gestion des différentes méthodes. On a pu voir enfin que la librairie FastAPI est de manière générale plus rapide que Flask mais aussi qu'elle permet de gérer facilement les requêtes asynchrones.

### b. Évaluation

Dans cette partie, nous allons présenter l'évaluation du module FastAPI.

#### **Contexte**

Pour cette évaluation, nous allons nous placer dans la peau d'une entreprise qui crée des questionnaires via une application pour Smartphone ou pour navigateur Web. Pour simplifier l'architecture de ces différents produits, l'entreprise veut mettre en place une API. Celle-ci a pour but d'interroger une base de données pour retourner une série de questions.



L'objectif de cette évaluation est donc de créer cette API.

#### **Les données**

Notre base de données est représentée par un fichier csv disponible à cette [adresse](https://dst-de.s3.eu-west-3.amazonaws.com/fastapi_fr/questions.csv).

Vous pouvez télécharger le jeu de données sur la machine en faisant:

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

On y retrouve les champs suivants:

* question: l'intitulé de la question
* subject: la catégorie de la question
* correct: la liste des réponses correctes
* use: le type de QCM pour lequel cette question est utilisée
* responseA: réponse A
* responseB: réponse B
* responseC: réponse C
* responseD: la réponse D (si elle existe)

Explorez ce jeu de données pour comprendre ces données

#### **L'API**

Sur l'application ou le navigateur Web, l'utilisateur doit pouvoir choisir un type de test (use) ainsi qu'une ou plusieurs catégories (subject). De plus, l'application peut produire des QCMs de 5, 10 ou 20 questions. L'API doit donc être en mesure de retourner ce nombre de questions. Comme l'application doit pouvoir générer de nombreux QCMs, les questions doivent être retournées dans un ordre aléatoire: ainsi, une requête avec les mêmes paramètres pourra retourner des questions différentes.

Les utilisateurs devant avoir créé un compte, il faut que nous soyons en mesure de vérifier leurs identifiants. Pour l'instant l'API utilise une authentification basique, à base de nom d'utilisateur et de mot de passe: la chaîne de caractères contenant Basic username:password devra être passée dans l'en-tête Authorization (en théorie, cette chaîne de caractère devrait être encodée mais pour simplifier l'exercice, on peut choisir de ne pas l'encoder)

Pour les identifiants, on pourra utiliser le dictionnaire suivant:

{

"alice": "wonderland",

"bob": "builder",

"clementine": "mandarine"

}

L'API devra aussi implémenter un point de terminaison pour vérifier que l'API est bien fonctionnelle. Une autre fonctionnalité doit pouvoir permettre à un utilisateur admin dont le mot de passe est 4dm1N de créer une nouvelle question.

Enfin, elle devra être largement documentée et devra renvoyer des erreurs lorsque celle-ci est mal appelée.

#### **Rendus**

Les attendus sont un ou plusieurs fichiers Python contenant le code de l'API. On pourra aussi fournir un fichier requirements.txt listant les librairies à installer et un fichier contenant les instructions pour lancer l'API. Enfin, vous pouvez fournir un document expliquant les choix d'architecture effectués.