PROG2 - API | TD FastAPI + Postman

Objectifs académiques :

- Prendre en main le framework Fast API pour la création d'un webservice (API REST)
- Savoir manipuler les requêtes HTTP et les réponses HTTP à travers le service

TD1 : Prise en main de Fast API | 4 & 7 juillet 2025 - Groupe K1-K2-K3-K4-K5

Sur votre IDE (IntelliJ) :

1. Cloner le repository : https://github.com/hei-ryan/hello-world-api-python.git

2. Installer les dépendances nécessaires (recommandées automatiquement par votre IDE normalement).
   Si ce n'est pas le cas, faire manuellement l'installation des dépendances avec :
   pip install uvicorn fastapi

3. Lancer le serveur, soit à travers la commande :
   a. Si sur Linux ou MacOS : "./venv/bin/uvicorn main:app --reload" soit à travers la commande "uvicorn main:app --reload".

   b. Si sur Windows : "python .\venv\bin\uvicorn main:app –reload"

      En cas d'erreur, vous pouvez essayer de créer l'environnement virtuel avec la commande "python -m venv venv"

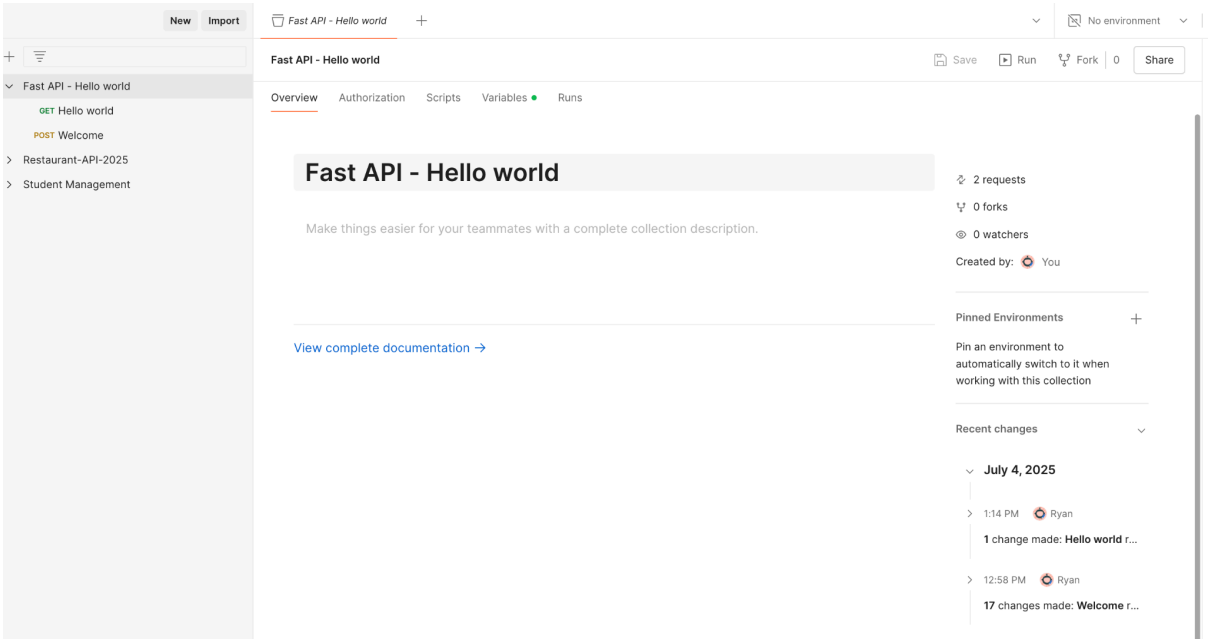
      Si la commande a été exécuté avec succès, votre terminal devrait afficher les messages suivant (ou similaire) :

```
(.venv) afryan@MacBook-Pro-de-Ryan hello-word-api-python % ./venv/bin/uvicorn main:app --reload

INFO:     Will watch for changes in these directories: ['/Users/afryan/IdeaProjects/hello-word-api-python']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [28245] using StatReload
INFO:     Started server process [28247]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Sur Postman :

1. Importer le fichier se trouvant dans /docs du projet cloné intitulé **Fast API - Hello world.postman_collection.json** en tant que collection Postman, et suivre les étapes. Vous devez tomber sur un écran similaire une fois que l'opération a été effectuée avec succès :



2. Lancez la requête GET Hello World. Quel résultat avez-vous obtenu ?

3. Lancez la requête POST Welcome. Quel résultat avez-vous obtenu ?

TD2 : Manipulation des requêtes avec Fast API | 9 & 10 juillet 2025 - Groupe K1-K2-K3-K4-K5

Documentation sur FastAPI : https://devdocs.io/fastapi/

L'objectif est de pouvoir manipuler la récupération des informations données par le client (comme Postman ou Navigateur ou tout autre application) et retourner cela dans le corps de la réponse retournée par le serveur (FastAPI).

a) **Récupération des données depuis l'URL :**

Il faut mettre en argument de la fonction les paramètres qu'on veut récupérer à travers l'URL.
Par exemple, pour l'URL suivant :
http://localhost:8000/hello?name=Ryan&is_teacher=true
On peut constater par définition que **name** c'est le **premier paramètre** car il se trouve après le caractère "?", et sa valeur ici est de type chaîne de caractère "Ryan". Et on retrouve le second paramètre **isTeacher** avec la valeur booléenne **true.**

Dans la fonction **def read_hello(request: Request)** qui permet d'exécuter les instructions lorsque le client invoque GET /hello, il suffit de mettre les deux paramètres de requête ainsi que leur type pour récupérer leurs valeurs.

Travail à faire 1 :

Tout d'abord, modifier la fonction **def read_hello** en conséquence afin de pouvoir récupérer ces données et modifier la réponse de sorte à ce que le message retourné comme ceci :
Si name=Ryan et is_teacher=true, alors le message retourné est "Hello Teacher Ryan !"
Si name=Rakoto et is_teacher=false, alors le message retourné est "Hello Rakoto !"

Indice pour la récupération, la fonction doit être modifiée comme suit mais avec les bons paramètres : **def read_hello(param_name: type)**

Testez sur Postman en ajoutant les paramètres de requête sur votre URL. Vous allez devoir dupliquer la requête GET Hello World, car vous avez deux exemples à tester, sur la première requête, vous allez ajouter les paramètres de requête name=Ryan et is_teacher=true et dans la deuxième requête que vous aurez dupliquée, vous allez ajouter les paramètres name=Rakoto et is_teacher=false.

Travail à faire 2 :

A - Veuillez noter que si vous supprimez les paramètres de requêtes préalablement ajoutées (name et is_teacher), une réponse 422 sera retournée indiquant que ces paramètres sont obligatoires, le travail consiste ainsi à les rendre facultatifs. Dans le cas où aucun des deux paramètres ne sont fournis, on reste sur le GET /hello initial,

modifiez le comportement de la fonction de sorte à retourner le message initial “Hello world”. Faites en sorte de ne pas casser le comportement lorsque les deux paramètres sont fournis.

B - Pour le cas où un seul paramètre est fourni, il faut faire en sorte d’ajouter des valeurs par défaut pour ceux qui n’ont pas été fournis.

Indice pour la récupération, la fonction doit être modifiée comme suit mais avec les bons paramètres : **def read_hello (name: str = “valeur par défaut”)**
“name” est ici le paramètre de type chaîne de caractère, (comme un String en java) et “valeur par défaut” est la valeur par défaut.

Premier exemple, si le client ne fournit que le paramètre name, alors il faut par défaut mettre la valeur de is_teacher à false, (et gérer le comportement équivalent) même si ce n’est pas présent dans l’URL. L’URL en question va être par exemple GET http://localhost:8000/hello?name=Ryan

Deuxième exemple, si le client ne fournit ne que le paramètre is_teacher, alors il faut par défaut mettre la valeur de name à “Non fourni”, (et gérer le comportement équivalent) même si ce n’est pas présent dans l’URL. L’URL en question va être par exemple GET http://localhost:8000/hello?is_teacher=true

Attention ! Il faut maintenir le comportement indiqué dans **A,** c’est à dire on ne considère les valeurs par défaut, uniquement lorsque l’un des deux paramètres sont fournis, car dans le cas où aucun n’est fourni, il ne faut pas

Testez sur Postman en ajoutant de nouvelles requêtes afin de vérifier que tout est bon.

b) **Récupération des données depuis les entêtes et le corps de la requête :**

Récupération en-tête :

Les entêtes de la requête peuvent être récupérées à travers request.headers.get(“<nom de l’entête>”), où request est un argument de type Request de la fonction invoquée lors de l’appel du chemin.

Par exemple, comme indiqué dans le code fourni, request.headers.get(“Accept”), permet de récupérer l’entête Accept envoyé par le client.

Récupération du corps de la requête :

Regarder l’exemple fourni selon **@app.post(“/welcome”)**, et essayer d’interpréter le code en vous appuyant sur la documentation présente ici : https://devdocs.io/fastapi/

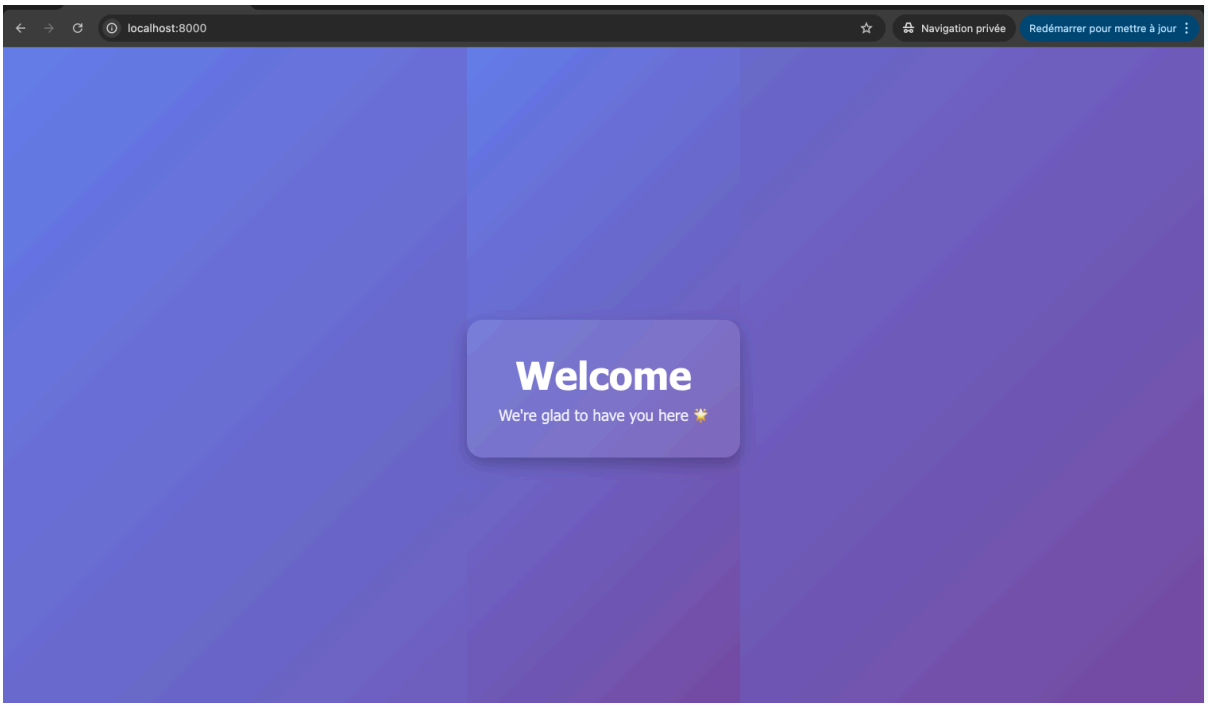
Travail à faire :

A - Créer un nouveau chemin accessible à travers PUT /top-secret. Une fois créé, vérifiez que les en-têtes envoyés par le client, comportent l'en-tête "Authorization" avec comme valeur "my-secret-key". Si aucune valeur n'est fournie, ou encore, si la valeur fournie ne correspond pas à "my-secret-key", alors il faut retourner une réponse avec un code de statut 403 FORBIDDEN et dans le corps de la réponse, indiquer la valeur qui a été fournie, mais qui n'a pas a été attendue.

B - Ajouter un corps de requête qui attend un JSON contenant les attributs "secret_code" de type int, qui doit comporter 4 chiffres. Si ce qui est fourni par le client est inférieur ou supérieur à 4 chiffres, il faut retourner une réponse 400 BAD_REQUEST avec comme message indiquant que le code fourni n'est pas à 4 chiffres. Si tout se passe, retourner le code à 4 chiffres fourni par le client.

TD3 : Suite manipulation des requêtes et réponses avec Fast API | Semaine 14 juillet 2025 - Groupe K1-K2-K3-K4-K5

L'objectif est de créer un petit API qui permet de gérer un calendrier manipulant essentiellement des événements à une date donnée et heure donnée (voir exemple de calendar.google.com). Le code de base se trouve se le repository suivant https://github.com/hei-ryan/python-api-mini-calendar.git et devrait vous permettre entre autres d'avoir la page HTML suivante qui s'affiche si vous ouvrez depuis un navigateur l'adresse http://127.0.0.1:8000 ou http://127.0.0.1:8000/ une fois que le serveur uvicorn tourne correctement.



A - Considérons le code suivant qui permet d'obtenir ce résultat :

```
1 from fastapi import FastAPI
2 from starlette.responses import Response
3
4 app = FastAPI()
5
6 @app.get("/")
7 def root():
8   with open("welcome.html", "r", encoding="utf-8") as file:
9       html_content = file.read()
10         return   Response(content=html_content,   status_code=200,
media_type="text/html")
```

A-1) Interprétez chaque ligne (de 1 à 10) de ce code pour comprendre ce qu'il effectue selon le résultat obtenu.

A-2) Modifier le code pour vérifier que la valeur de l'entête Accept (qui veut dire la réponse attendue par le client), doit être soit "text/html", soit "text/plain", sinon une réponse 400 contenant un message au format JSON indiquant que le type de format attendu n'est pas supporté et seul "text/html" ou "text/plain" le sont.

A-3) Modifier le code afin de vérifier que l'en-tête de la requête contient l'en-tête spécifique "x-api-key" (au lieu de "Authorization") avec comme valeur "12345678", avant de retourner la page web comme réponse, sinon il faut retourner une réponse 403 contenant un message au format JSON indiquant que la clé API fournie n'est pas reconnue.

B - Le code suivant, suite du code précédent, permet de retourner une erreur 404 personnalisée pour tous les chemins qui n'ont pas été défini directement dans le serveur.

```
1 import json
2 from fastapi import FastAPI
3 from starlette.responses import Response
4
5 app = FastAPI()
6
7
8 #@app.get("/") et tous les autres chemins spécifiques ici …
9 #def root():
10 # etc ...
11 # etc ...
12 # etc …
13
14
15 @app.get("/{full_path:path}")
16 def catch_all(full_path: str):
17     not_found_message = {"detail": f"Page '/{full_path}' not found"}
18         return   Response(content=json.dumps(not_found_message), status_code=404, media_type="application/json")
```

Le premier changement réside dans la ligne 1, où nous avons importé la fonction json.dumps() qui nous permet de dire explicitement au serveur que le contenu qu'on veut retourner dans la réponse est de type JSON.

Le deuxième changement réside dans la ligne 11, où nous avons utilisé une route (ou chemin) **tout à la fin** avec un chemin dynamique qui capture **tous les chemins restants**, indiqué par **{full_path:path}**

Le dernier changement réside dans la ligne 14, où tout d'abord nous constatons l'usage de json.dumps(not_found_message), pour expliciter la conversion de la chaîne de caractères (type str en python) en JSON dans le contenu de la réponse. Remarquez qu'on aurait pu directement utiliser JSONResponse au lieu de Response ici.

Pour tester que tout se passe bien, étant donné que le seul chemin ou route défini actuellement dans notre cas est “/”, donc tout ce qui va être mis après le séparateur “/” devrait retourner une réponse 404 avec un contenu “Page /xxx not found” où xxx est ici le chemin que vous aurez émis depuis le client (Postman ou navigateur). Par exemple, si vous avez essayé de faire http://localhost:8000/mon-chemin alors vous obtiendrez la réponse en JSON : “ {"detail": "Page '/mon-chemin’ not found"}, par contre si vous faites juste http://localhost:8000/ étant donné que ça a déjà été défini, alors ça va bien retourner la page web contenant le contenu du fichier welcome.html.

a) En utilisant les bases de codes fournis (et uniquement celles-ci), créez une nouvelle page web statique (et non dynamique) en vous inspirant de **welcome.html**, indiquant que au client que la ressource demandée n’a pas été trouvée ave'c un “404 NOT FOUND”, et modifier la réponse de la fonction **def catch_all** pour retourner cette page web statique au lieu de retourner une réponse au format JSON.

   NB : Page statique veut dire qu’elle va avoir le même contenu tout le temps pour juste dire NOT FOUND, mais pas qu’elle va dire quelle route ou chemin n’a pas été trouvée comme dans l’exemple sur la version en JSON.

b) Créer une nouvelle route (ou chemin) à travers GET /events, qui va permettre de récupérer au format JSON, les évènements enregistrés dans le serveur. Actuellement, nous allons enregistrer les évènements en mémoire vive dans une liste d’objets. La classe qui permet de spécifier un événement dans le calendrier est la suivante :

```
class EventModel(BaseModel):

    name: str

    description: str

    start_date: str

    end_date: str
```

   Pour créer la liste qui va sauvegarder les données, :

   - Il faut tout d’abord importer List depuis le module typing à travers la syntaxe : `from typing import List`
   - Créez la liste en l’initialisant avec une liste vide au début : `events_store: List[EventModel] = []`

     Ici, `events_store` est le nom de la liste que nous allons manipuler par la suite pour stocker les événements. C’est une liste qui va contenir des objets de type `EventModel` et qui est initialisée à une valeur par défaut vide `[]`

   Pour faciliter la conversion de l’objet EventModel en JSON, l'idéal est de créer directement une fonction qui va récupérer la liste `events_store`, et qui va la

serialiser[1] directement et que l'on pourra utiliser pour toute la suite. Voici la fonction en question que l'on va commenter :

```
def serialized_stored_events():
    events_converted = []
    for event in events_store:
        events_converted.append(event.model_dump())
    return events_converted
```

Dans la fonction `serialized_stored_events` donnée, c'est la fonction `model_dump()` de chaque objet EventModel de la liste `events_store` qui effectue la sérialisation.

Grâce à cette fonction `serialized_stored_events` modulaire, le chemin GET /events devient tout simplement :

```
@app.get("/events")
def list_events():
    return {"events": serialized_stored_events()}
```

c) Créer une nouvelle route (ou chemin) à travers POST /events, qui va permettre d'ajouter une liste d'évènements, fournis à travers le corps de la requête par le client, dans la liste `events_store`. La réponse attendue est la liste des évènements contenus dans `events_store` avec les nouvelles valeurs ajoutées. Se référer à la liste des fonctions natives en python fournis à la fin de ce document pour la manipulation des listes si nécessaires.

d) Créer une nouvelle route (ou chemin) à travers PUT /events, qui va permettre de modifier un ou plusieurs événements fournis à travers le corps de la requête par le client et existants dans la liste `events_store`. La réponse attendue est la liste des évènements contenus dans `events_store` avec les nouvelles valeurs modifiées.

Dans le cas où l'objet JSON fourni n'existe pas encore dans la liste, alors il faut le créer (même comportement qu'un POST /events dans ce cas). On parle alors de ***requête idempotente***[2].

---

[1] La **sérialisation** consiste à **transformer ou convertir un objet** (de n'importe quel langage) en un **format transférable** ou **stockable**, comme du **JSON**, **XML**, **bytes**, etc.

[2] Une **requête idempotente** est une opération HTTP qui produit **le même résultat**, **peu importe combien de fois elle est répétée**.

Méthodes natives (ou existantes par défaut) des listes Python

Méthode	Description
`append(x)`	Ajoute un élément à la fin
`extend(iterable)`	Ajoute plusieurs éléments
`insert(i, x)`	Insère un élément à une position donnée
`remove(x)`	Supprime la **1re occurrence** de `x`
`pop(i)`	Supprime et retourne l'élément à l'indice `i` (ou le dernier si omis)
`clear()`	Supprime tous les éléments
`index(x)`	Donne l'indice de la 1re occurrence de `x`
`count(x)`	Compte le nombre d'occurrences de `x`
`sort()`	Trie la liste (sur place)
`reverse()`	Inverse la liste (sur place)
`copy()`	Fait une copie superficielle de la liste