FastAPI - Passer des données à une API

60 minutes Normal

DataScientest • com

APIs avec FastAPI

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

```
2
3  api = FastAPI(
4      title='My API'
5  )
6
7  @api.get('/')
8  def get_index():
9      return {'data': 'hello world'}
10
```

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

```
1  uvicorn main:api --reload
2
```

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/

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

```
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'
        }

```

FastAPI fournit de plus de nombreuses façons de contrôler le type des arguments fourni 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/

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

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

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

```

retournera le résultat.

> Inversez l'ordre de définition des fonctions de manières à avoir `get_item_default` en premier et relancez les requêtes précédentes.

Exercice d'application

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 reservirons dans la suite.

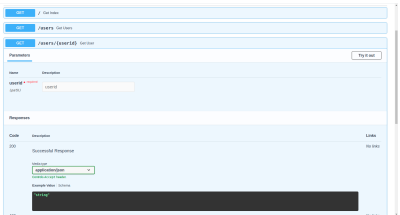
Show / Hide solution

```
from fastapi import FastAPI

users_db = [
    {
        'user_id': 1,
        'name': 'Alice',
        'subscription': 'free 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.
        return user
    except IndexError:
        return {}

@api.get('/users/{userid:int}/name')
def get_user_name(userid):
    try:
        user = list(filter(lambda x: x.
        return {'name': user['name']}
    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

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/?argu

```

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

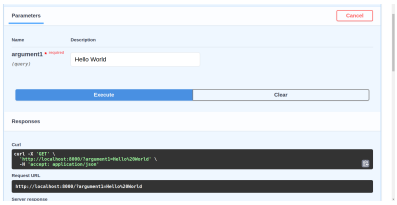
```
curl -X GET -i http://127.0.0.1:8000/

```

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



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

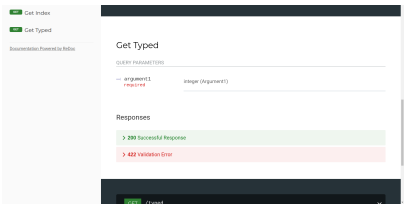
```
curl -X GET -i http://127.0.0.1:8000/typed

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

```

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

```
{
  "detail": [
    {
      "loc": ["query", "argument1"],
      "msg": "value is not a valid inte
      "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.

```
2
3  @api.get('/addition')
4  def get_addition(a: int, b: Optional[ir
5      if b:
6          result = a + b
7      else:
8          result = a + 1
9      return {
10         'addition_result': result
11     }
```

> 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

```
1  from pydantic import BaseModel
2  from typing import Optional
3
4  class Item(BaseModel):
5      itemid: int
6      description: str
7      owner: Optional[str] = None
8
```

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

```
1  @api.post('/item')
2  def post_item(item: Item):
3      return {
4          'itemid': item.itemid
5      }
6
```

> 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 '{
```

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`.

Show / Hide solution

```
@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

Show / Hide solution

```
@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

```

```
    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.

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`.

Show / Hide solution

```
    subscription: str

@api.put('/users')
def put_users(user: User):
    new_id = max(users_db, key=lambda u
    new_user = {
        'user_id': new_id + 1,
        'name': user.name,
        'subscription': user.subscripti
    }
    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('use
            )[0]

        users_db.remove(old_user)

        old_user['name'] = user.name
        old_user['subscription'] = user

        users_db.append(old_user)
        return old_user

    except IndexError:
```

```
40  def delete_users(userid):
41      try:
42          old_user = list(
43              filter(lambda x: x.get('use
44              )[0]
45
46          users_db.remove(old_user)
47          return {
48              'userid': userid,
49              'deleted': True
50              }
51          except IndexError:
52           return {}
53
```

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:

```
1  from fastapi import Header
2
3  @api.get('/headers')
4  def get_headers(user_agent=Header(None))
5      return {
6          'User-Agent': user_agent
7      }
8
```

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

```
1  curl -X GET -i http://127.0.0.1:8000/heade
2
```

La réponse devrait être:

```
1  { "User-Agent": "curl/7.68.0" }
2
```

> 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

Robin
BIRON

Validated