Paramètres de chemin

Vous pouvez déclarer des "paramètres" ou "variables" de chemin avec la même syntaxe que celle utilisée par le formatage de chaîne Python :

```
{!../../../docs_src/path_params/tutorial001.py!}
```

La valeur du paramètre `item_id` sera transmise à la fonction dans l'argument `item_id`.

Donc, si vous exécutez cet exemple et allez sur http://127.0.0.1:8000/items/foo, vous verrez comme réponse :

```
{"item_id":"foo"}
```

Paramètres de chemin typés

Vous pouvez déclarer le type d'un paramètre de chemin dans la fonction, en utilisant les annotations de type Python :

```
{!../../../docs_src/path_params/tutorial002.py!}
```

Ici, `item_id` est déclaré comme `int`.

!!! hint "Astuce" Ceci vous permettra d'obtenir des fonctionnalités de l'éditeur dans votre fonction, telles que des vérifications d'erreur, de l'auto-complétion, etc.

Conversion de données

Si vous exécutez cet exemple et allez sur http://127.0.0.1:8000/items/3, vous aurez comme réponse :

```
{"item_id":3}
```

!!! hint "Astuce" Comme vous l'avez remarqué, la valeur reçue par la fonction (et renvoyée ensuite) est `3`, en tant qu'entier (`int`) Python, pas la chaîne de caractères (`string`) `"3"`.

```
Grâce aux déclarations de types, **FastAPI** fournit du
<abbr title="conversion de la chaîne de caractères venant de la requête HTTP en
données Python">"parsing"</abbr> automatique.
```

Validation de données

Si vous allez sur http://127.0.0.1:8000/items/foo, vous aurez une belle erreur HTTP :

```
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
```

```
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

car le paramètre de chemin `item_id` possède comme valeur `"foo"`, qui ne peut pas être convertie en entier (`int`).

La même erreur se produira si vous passez un nombre flottant (`float`) et non un entier, comme ici http://127.0.0.1:8000/items/4.2.

!!! hint "Astuce" Donc, avec ces mêmes déclarations de type Python, **FastAPI** vous fournit de la validation de données.

```
Notez que l'erreur mentionne le point exact où la validation n'a pas réussi.

Ce qui est incroyablement utile au moment de développer et débugger du code qui
interagit avec votre API.
```

Documentation

Et quand vous vous rendez sur http://127.0.0.1:8000/docs, vous verrez la documentation générée automatiquement et interactive :

!!! info À nouveau, en utilisant uniquement les déclarations de type Python, **FastAPI** vous fournit automatiquement une documentation interactive (via Swagger UI).

```
On voit bien dans la documentation que `item_id` est déclaré comme entier.
```

Les avantages d'avoir une documentation basée sur une norme, et la documentation alternative.

Le schéma généré suivant la norme OpenAPI, il existe de nombreux outils compatibles.

Grâce à cela, **FastAPI** lui-même fournit une documentation alternative (utilisant ReDoc), qui peut être lue sur http://127.0.0.1:8000/redoc :

De la même façon, il existe bien d'autres outils compatibles, y compris des outils de génération de code pour de nombreux langages.

Pydantic

Toute la validation de données est effectué en arrière-plan avec Pydantic, dont vous bénéficierez de tous les avantages. Vous savez donc que vous êtes entre de bonnes mains.

L'ordre importe

Quand vous créez des *fonctions de chemins*, vous pouvez vous retrouver dans une situation où vous avez un chemin fixe.

Tel que `/users/me`, disons pour récupérer les données sur l'utilisateur actuel.

Et vous avez un second chemin : `/users/{user_id}` pour récupérer de la donnée sur un utilisateur spécifique grâce à son identifiant d'utilisateur

Les *fonctions de chemin* étant évaluées dans l'ordre, il faut s'assurer que la fonction correspondant à `/users/me` est déclarée avant celle de `/users/{user_id}` :

```
{!../../../docs_src/path_params/tutorial003.py!}
```

Sinon, le chemin `/users/{user_id}` correspondrait aussi à `/users/me`, la fonction "croyant" qu'elle a reçu un paramètre `user_id` avec pour valeur `"me"`.

Valeurs prédéfinies

Si vous avez une *fonction de chemin* qui reçoit un *paramètre de chemin*, mais que vous voulez que les valeurs possibles des paramètres soient prédéfinies, vous pouvez utiliser les `Enum` de Python.

Création d'un `Enum`

Importez `Enum` et créez une sous-classe qui hérite de `str` et `Enum`.

En héritant de `str` la documentation sera capable de savoir que les valeurs doivent être de type `string` et pourra donc afficher cette `Enum` correctement.

Créez ensuite des attributs de classe avec des valeurs fixes, qui seront les valeurs autorisées pour cette énumération.

```
{!../../../docs_src/path_params/tutorial005.py!}
```

!!! info Les énumérations (ou enums) sont disponibles en Python depuis la version 3.4.

!!! tip "Astuce" Pour ceux qui se demandent, "AlexNet", "ResNet", et "LeNet" sont juste des noms de modèles de Machine Learning.

Déclarer un paramètre de chemin

Créez ensuite un *paramètre de chemin* avec une annotation de type désignant l'énumération créée précédemment (`ModelName`) :

```
{!../../../docs_src/path_params/tutorial005.py!}
```

Documentation

Les valeurs disponibles pour le *paramètre de chemin* sont bien prédéfinies, la documentation les affiche correctement :

Manipuler les *énumérations* Python

La valeur du *paramètre de chemin* sera un des "membres" de l'énumération.

Comparer les *membres d'énumération*

Vous pouvez comparer ce paramètre avec les membres de votre énumération `ModelName` :

```
{!../../../docs_src/path_params/tutorial005.py!}
```

Récupérer la *valeur de l'énumération*

Vous pouvez obtenir la valeur réel d'un membre (une chaîne de caractères ici), avec `model_name.value`, ou en général, `votre_membre_d'enum.value` :

```
{!../../../docs_src/path_params/tutorial005.py!}
```

!!! tip "Astuce" Vous pouvez aussi accéder la valeur `"lenet"` avec `ModelName.lenet.value`.

Retourner des *membres d'énumération*

Vous pouvez retourner des *membres d'énumération* dans vos *fonctions de chemin*, même imbriquée dans un JSON (e.g. un `dict`).

Ils seront convertis vers leurs valeurs correspondantes (chaînes de caractères ici) avant d'être transmis au client :

```
{!../../../docs_src/path_params/tutorial005.py!}
```

Le client recevra une réponse JSON comme celle-ci :

```
{
  "model_name": "alexnet",
  "message": "Deep Learning FTW!"
}
```

Paramètres de chemin contenant des chemins

Disons que vous avez une *fonction de chemin* liée au chemin `/files/{file_path}`.

Mais que `file_path` lui-même doit contenir un *chemin*, comme `home/johndoe/myfile.txt` par exemple.

Donc, l'URL pour ce fichier pourrait être : `/files/home/johndoe/myfile.txt`.

Support d'OpenAPI

OpenAPI ne supporte pas de manière de déclarer un paramètre de chemin contenant un *chemin*, cela pouvant causer des scénarios difficiles à tester et définir.

Néanmoins, cela reste faisable dans **FastAPI**, via les outils internes de Starlette.

Et la documentation fonctionne quand même, bien qu'aucune section ne soit ajoutée pour dire que la paramètre devrait contenir un *chemin*.

Convertisseur de *chemin*

En utilisant une option de Starlette directement, vous pouvez déclarer un *paramètre de chemin* contenant un *chemin* avec une URL comme :

```
/files/{file_path:path}
```

Dans ce cas, le nom du paramètre est `file_path`, et la dernière partie, `:path`, indique à Starlette que le paramètre devrait correspondre à un *chemin*.

Vous pouvez donc l'utilisez comme tel :

```
{!../../../docs_src/path_params/tutorial004.py!}
```

!!! tip "Astuce" Vous pourriez avoir besoin que le paramètre contienne `/home/johndoe/myfile.txt`, avec un slash au début (`/`).

```
Dans ce cas, l'URL serait : `/files//home/johndoe/myfile.txt`, avec un double slash
(`//`) entre `files` et `home`.
```

Récapitulatif

Avec **FastAPI**, en utilisant les déclarations de type rapides, intuitives et standards de Python, vous bénéficiez de :

- Support de l'éditeur : vérification d'erreurs, auto-complétion, etc.
- "Parsing" de données.
- Validation de données.
- Annotations d'API et documentation automatique.

Et vous n'avez besoin de le déclarer qu'une fois.

C'est probablement l'avantage visible principal de **FastAPI** comparé aux autres *frameworks* (outre les performances pures).