Corps de la requête

Quand vous avez besoin d'envoyer de la donnée depuis un client (comme un navigateur) vers votre API, vous l'envoyez en tant que **corps de requête**.

Le corps d'une **requête** est de la donnée envoyée par le client à votre API. Le corps d'une **réponse** est la donnée envoyée par votre API au client.

Votre API aura presque toujours à envoyer un corps de **réponse**. Mais un client n'a pas toujours à envoyer un corps de **requête**.

Pour déclarer un corps de **requête**, on utilise les modèles de Pydantic en profitant de tous leurs avantages et fonctionnalités.

!!! info Pour envoyer de la donnée, vous devriez utiliser : `POST` (le plus populaire), `PUT`, `DELETE` ou `PATCH`.

```
Envoyer un corps dans une requête `GET` a un comportement non défini dans les spécifications, cela est néanmoins supporté par **FastAPI**, seulement pour des cas d'utilisation très complexes/extrêmes.

Ceci étant découragé, la documentation interactive générée par Swagger UI ne montrera pas de documentation pour le corps d'une requête `GET`, et les proxys intermédiaires risquent de ne pas le supporter.
```

Importez le `BaseModel` de Pydantic

Commencez par importer la classe `BaseModel` du module `pydantic` :

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

Créez votre modèle de données

Déclarez ensuite votre modèle de données en tant que classe qui hérite de `BaseModel`.

Utilisez les types Python standard pour tous les attributs :

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

Tout comme pour la déclaration de paramètres de requête, quand un attribut de modèle a une valeur par défaut, il n'est pas nécessaire. Sinon, cet attribut doit être renseigné dans le corps de la requête. Pour rendre ce champ optionnel simplement, utilisez `None` comme valeur par défaut.

Par exemple, le modèle ci-dessus déclare un "objet" JSON (ou `dict` Python) tel que :

```
{
    "name": "Foo",
    "description": "An optional description",
    "price": 45.2,
    "tax": 3.5
}
```

... `description` et `tax` étant des attributs optionnels (avec `None` comme valeur par défaut), cet "objet" JSON serait aussi valide :

```
{
    "name": "Foo",
    "price": 45.2
}
```

Déclarez-le comme paramètre

Pour l'ajouter à votre *opération de chemin*, déclarez-le comme vous déclareriez des paramètres de chemin ou de requête :

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

...et déclarez que son type est le modèle que vous avez créé : `Item`.

Résultats

En utilisant uniquement les déclarations de type Python, **FastAPI** réussit à :

- Lire le contenu de la requête en tant que JSON.
- Convertir les types correspondants (si nécessaire).
- Valider la donnée.
    - Si la donnée est invalide, une erreur propre et claire sera renvoyée, indiquant exactement où était la donnée incorrecte.
- Passer la donnée reçue dans le paramètre `item`.
    - Ce paramètre ayant été déclaré dans la fonction comme étant de type `Item`, vous aurez aussi tout le support offert par l'éditeur (auto-complétion, etc.) pour tous les attributs de ce paramètre et les types de ces attributs.
- Générer des définitions JSON Schema pour votre modèle, qui peuvent être utilisées où vous en avez besoin dans votre projet ensuite.
- Ces schémas participeront à la constitution du schéma généré OpenAPI, et seront donc utilisés par les documentations automatiquement générées.

Documentation automatique

Les schémas JSON de vos modèles seront intégrés au schéma OpenAPI global de votre application, et seront donc affichés dans la documentation interactive de l'API :

Et seront aussi utilisés dans chaque *opération de chemin* de la documentation utilisant ces modèles :

Support de l'éditeur

Dans votre éditeur, vous aurez des annotations de types et de l'auto-complétion partout dans votre fonction (ce qui n'aurait pas été le cas si vous aviez utilisé un classique `dict` plutôt qu'un modèle Pydantic) :

Et vous obtenez aussi de la vérification d'erreur pour les opérations incorrectes de types :

Ce n'est pas un hasard, ce framework entier a été bati avec ce design comme objectif.

Et cela a été rigoureusement testé durant la phase de design, avant toute implémentation, pour s'assurer que cela fonctionnerait avec tous les éditeurs.

Des changements sur Pydantic ont même été faits pour supporter cela.

Les captures d'écrans précédentes ont été prises sur Visual Studio Code.

Mais vous auriez le même support de l'éditeur avec PyCharm et la majorité des autres éditeurs de code Python.

!!! tip "Astuce" Si vous utilisez PyCharm comme éditeur, vous pouvez utiliser le Plugin PyCharm.

```
Ce qui améliore le support pour les modèles Pydantic avec :

* de l'auto-complétion
* des vérifications de type
* du "refactoring" (ou remaniement de code)
* de la recherche
* de l'inspection
```

Utilisez le modèle

Dans la fonction, vous pouvez accéder à tous les attributs de l'objet du modèle directement :

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

Corps de la requête + paramètres de chemin

Vous pouvez déclarer des paramètres de chemin et un corps de requête pour la même *opération de chemin*.

FastAPI est capable de reconnaître que les paramètres de la fonction qui correspondent aux paramètres de chemin doivent être **récupérés depuis le chemin**, et que les paramètres de fonctions déclarés comme modèles Pydantic devraient être **récupérés depuis le corps de la requête**.

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

Corps de la requête + paramètres de chemin et de requête

Vous pouvez aussi déclarer un **corps**, et des paramètres de **chemin** et de **requête** dans la même *opération de chemin*.

FastAPI saura reconnaître chacun d'entre eux et récupérer la bonne donnée au bon endroit.

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

Les paramètres de la fonction seront reconnus comme tel :

- Si le paramètre est aussi déclaré dans le **chemin**, il sera utilisé comme paramètre de chemin.
- Si le paramètre est d'un **type singulier** (comme `int`, `float`, `str`, `bool`, etc.), il sera interprété comme un paramètre de **requête**.
- Si le paramètre est déclaré comme ayant pour type un **modèle Pydantic**, il sera interprété comme faisant partie du **corps** de la requête.

!!! note **FastAPI** saura que la valeur de `q` n'est pas requise grâce à la valeur par défaut `=None`.

```
Le type `Optional` dans `Optional[str]` n'est pas utilisé par **FastAPI**, mais sera utile à votre éditeur pour améliorer le support offert par ce dernier et détecter plus facilement des erreurs de type.
```

Sans Pydantic

Si vous ne voulez pas utiliser des modèles Pydantic, vous pouvez aussi utiliser des paramètres de **Corps**. Pour cela, allez voir la partie de la documentation sur Corps de la requête - Paramètres multiples{.internal-link target=_blank}.