

Fonctionnalités

Fonctionnalités de FastAPI

FastAPI vous offre ceci:

Basé sur des standards ouverts

- [OpenAPI](#) pour la création d'API, incluant la déclaration de `path operations`, paramètres, corps de requêtes, sécurité, etc.
- Documentation automatique des modèles de données avec [JSON Schema](#) (comme OpenAPI est aussi basée sur JSON Schema).
- Conçue avec ces standards après une analyse méticuleuse. Plutôt qu'en rajoutant des surcouches après coup.
- Cela permet d'utiliser de la **génération automatique de code client** dans beaucoup de langages.

Documentation automatique

Documentation d'API interactive et interface web d'exploration. Comme le framework est basé sur OpenAPI, de nombreuses options sont disponibles. Deux d'entre-elles sont incluses par défaut.

- [Swagger UI](#), propose une documentation interactive. Vous permet de directement tester l'API depuis votre navigateur.

Fast API - Swagger UI

127.0.0.1:8000/docs

Fast API 0.1.0 OAS3

/openapi.json

default

GET / Read Root Get

GET /items/{item_id} Read Item Get

PUT /items/{item_id} Save Item Put

Parameters Try it out

Name	Description
item_id * required integer (path)	

Request body required application/json

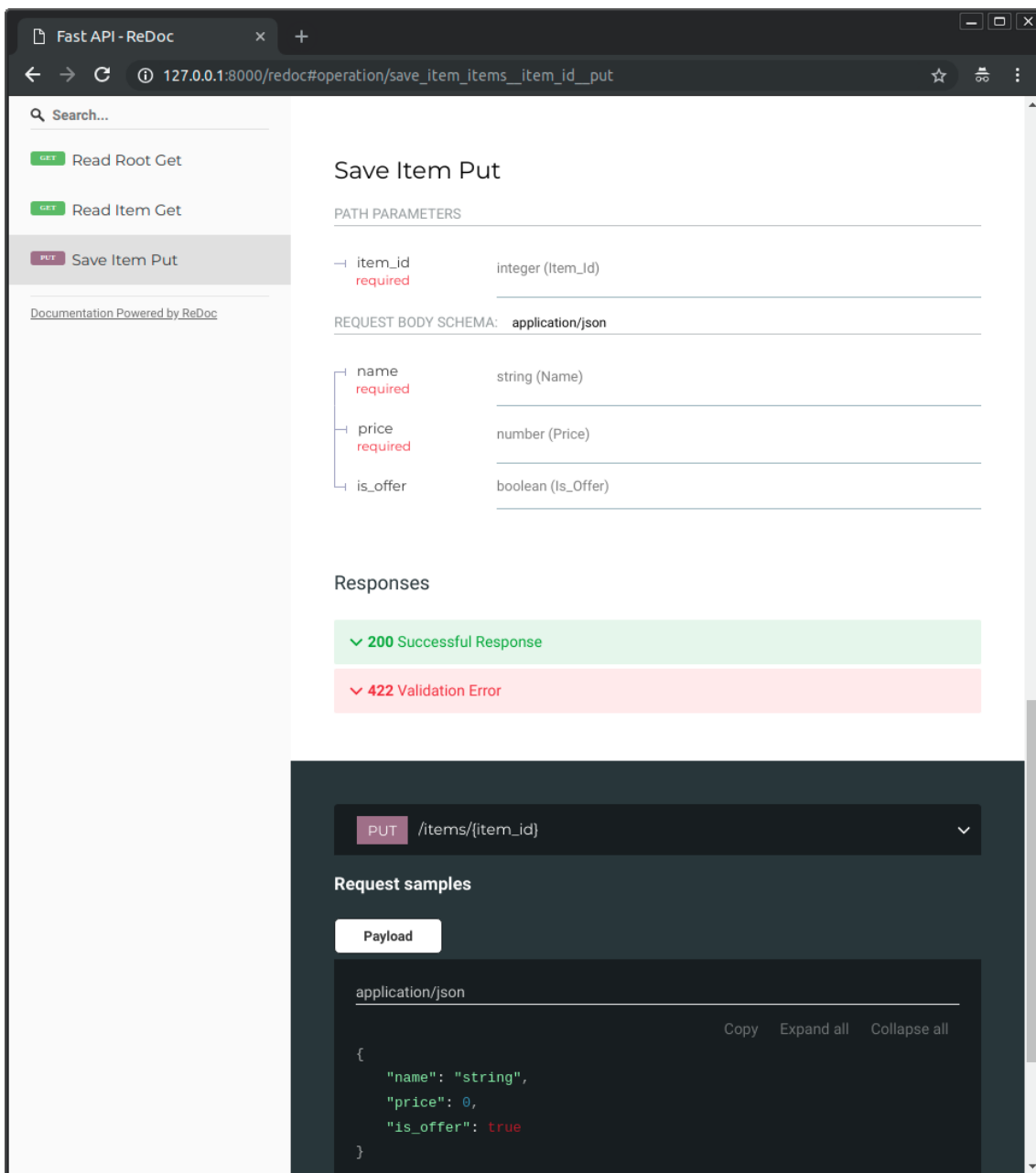
Example Value | Schema

```
{  
  "name": "string",  
  "price": 0,  
  "is_offer": true  
}
```

Responses

Code	Description	Links
200	Successful Response	No links

- Une autre documentation d'API est fournie par [ReDoc](#).



Faite en python moderne

Tout est basé sur la déclaration de type standard de **Python 3.6** (grâce à Pydantic). Pas de nouvelles syntaxes à apprendre. Juste du Python standard et moderne.

Si vous souhaitez un rappel de 2 minutes sur l'utilisation des types en Python (même si vous ne comptez pas utiliser FastAPI), jetez un oeil au tutoriel suivant: [Python Types](#) (internal-link target=_blank).

Vous écrivez du python standard avec des annotations de types:

```
from datetime import date

from pydantic import BaseModel
```

```
# Déclare une variable comme étant une str
# et profitez de l'aide de votre IDE dans cette fonction
def main(user_id: str):
    return user_id

# Un modèle Pydantic
class User(BaseModel):
    id: int
    name: str
    joined: date
```

Qui peuvent ensuite être utilisés comme cela:

```
my_user: User = User(id=3, name="John Doe", joined="2018-07-19")

second_user_data = {
    "id": 4,
    "name": "Mary",
    "joined": "2018-11-30",
}

my_second_user: User = User(**second_user_data)
```

!!! info `**second_user_data` signifie:

Utilise les clés et valeurs du dictionnaire `second_user_data` directement comme des arguments clé-valeur. C'est équivalent à: `User(id=4, name="Mary", joined="2018-11-30")`

Support d'éditeurs

Tout le framework a été conçu pour être facile et intuitif d'utilisation, toutes les décisions de design ont été testées sur de nombreux éditeurs avant même de commencer le développement final afin d'assurer la meilleure expérience de développement possible.

Dans le dernier sondage effectué auprès de développeurs python il était clair que [la fonctionnalité la plus utilisée est "l'autocomplétion"](#).

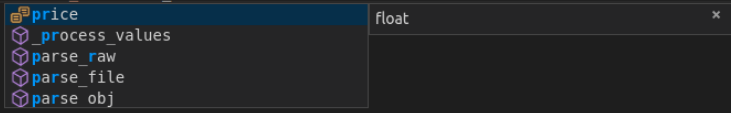
Tout le framework **FastAPI** a été conçu avec cela en tête. L'autocomplétion fonctionne partout.

Vous devrez rarement revenir à la documentation.

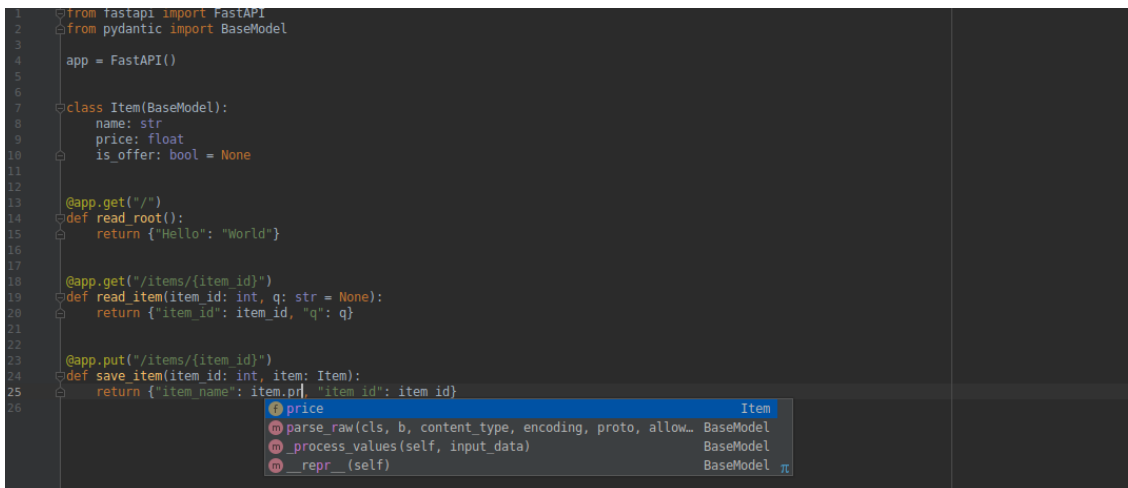
Voici comment votre éditeur peut vous aider:

- dans [Visual Studio Code](#):

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6
7 class Item(BaseModel):
8     name: str
9     price: float
10    is_offer: bool = None
11
12
13 @app.get("/")
14 def read_root():
15     return {"Hello": "World"}
16
17
18 @app.get("/items/{item_id}")
19 def read_item(item_id: int, q: str = None):
20     return {"item_id": item_id, "q": q}
21
22
23 @app.put("/items/{item_id}")
24 def save_item(item_id: int, item: Item):
25     return {"item_name": item.name, "item_id": item_id}
26
```



- dans [PyCharm](#):



Vous aurez des propositions de complétion que vous n'auriez jamais imaginées. Par exemple la clé `prix` dans le corps d'un document JSON (qui est peut-être imbriqué) venant d'une requête.

Plus jamais vous ne vous tromperez en tapant le nom d'une clé, vous ne ferez des aller-retour entre votre code et la documentation ou vous ne scrollerez de haut en bas afin d'enfin savoir si vous devez taper `username` ou `user_name`.

Court

Des **valeurs par défaut** sont définies pour tout, des configurations optionnelles sont présentes partout. Tous ces paramètres peuvent être ajustés afin de faire ce que vous voulez et définir l'API dont vous avez besoin.

Mais, **tout fonctionne** par défaut.

Validation

- Validation pour la plupart (ou tous?) les **types de données** Python incluant:
 - objets JSON (`dict`).
 - listes JSON (`list`) définissant des types d'éléments.
 - Champs String (`str`), définition de longueur minimum ou maximale.
 - Nombres (`int` , `float`) avec valeur minimale and maximale, etc.
- Validation pour des types plus exotiques, tel que:
 - URL.
 - Email.
 - UUID.
 - ...et autres.

Toutes les validations sont gérées par le bien établi et robuste **Pydantic**.

Sécurité et authentification

La sécurité et l'authentification sont intégrées. Sans aucun compromis avec les bases de données ou les modèles de données.

Tous les protocoles de sécurités sont définis dans OpenAPI, incluant:

- HTTP Basic.
- **OAuth2** (aussi avec **JWT tokens**). Jetez un oeil au tutoriel [OAuth2 avec JWT](#){.internal-link target=_blank}.
- Clés d'API dans:
 - Le header.
 - Les paramètres de requêtes.
 - Les cookies, etc.

Plus toutes les fonctionnalités de sécurités venant de Starlette (incluant les **cookies de sessions**).

Le tout conçu en composant réutilisable facilement intégrable à vos systèmes, data stores, base de données relationnelle ou NoSQL, etc.

Injection de dépendances

FastAPI contient un système simple mais extrêmement puissant d'Injection de Dépendances.

- Même les dépendances peuvent avoir des dépendances, créant une hiérarchie ou un "**graph**" de **dépendances**
- Tout est **automatiquement géré** par le framework
- Toutes les dépendances peuvent exiger des données d'une requête et **Augmenter les contraintes d'un path operation** et de la documentation automatique.
- **Validation automatique** même pour les paramètres de *path operation* définis dans les dépendances.
- Supporte les systèmes d'authentification d'utilisateurs complexes, les **connexions de base de données**, etc.
- **Aucun compromis** avec les bases de données, les frontends, etc. Mais une intégration facile avec n'importe lequel d'entre eux.

"Plug-ins" illimités

Ou, en d'autres termes, pas besoin d'eux, importez le code que vous voulez et utilisez le.

Tout intégration est conçue pour être si simple à utiliser (avec des dépendances) que vous pouvez créer un "plug-in" pour votre application en deux lignes de code utilisant la même syntaxe que celle de vos *path operations*

Testé

- 100% de couverture de test.
- 100% d'annotations de type dans le code.
- Utilisé dans des applications mises en production.

Fonctionnalités de Starlette

FastAPI est complètement compatible (et basé sur) [Starlette](#). Le code utilisant Starlette que vous ajouterez fonctionnera donc aussi.

En fait, `FastAPI` est un sous composant de `Starlette`. Donc, si vous savez déjà comment utiliser Starlette, la plupart des fonctionnalités fonctionneront de la même manière.

Avec **FastAPI** vous aurez toutes les fonctionnalités de **Starlette** (FastAPI est juste Starlette sous stéroïdes):

- Des performances vraiment impressionnantes. C'est l'[un des framework Python les plus rapide, à égalité avec NodeJS et GO](#).
- Le support des **WebSockets**.
- Le support de **GraphQL**.
- Les tâches d'arrière-plan.
- Des événements de démarrages et d'arrêt.
- Un client de test basé sur `request`
- **CORS**, GZip, Static Files, Streaming responses.
- Le support des **Sessions et Cookies**.
- Une couverture de test à 100 %.
- 100 % de la base de code avec des annotations de type.

Fonctionnalités de Pydantic

FastAPI est totalement compatible avec (et basé sur) [Pydantic](#). Le code utilisant Pydantic que vous ajouterez fonctionnera donc aussi.

Inclus des bibliothèques externes basées, aussi, sur Pydantic, servent d'ORMs, ODMs pour les bases de données.

Cela signifie aussi que, dans la plupart des cas, vous pouvez fournir l'objet reçu d'une requête **directement à la base de données**, comme tout est validé automatiquement.

Inversément, dans la plupart des cas vous pourrez juste envoyer l'objet récupéré de la base de données **directement au client**

Avec **FastAPI** vous aurez toutes les fonctionnalités de **Pydantic** (comme FastAPI est basé sur Pydantic pour toutes les manipulations de données):

- **Pas de prise de tête:**
 - Pas de nouveau langage de définition de schéma à apprendre.
 - Si vous connaissez le typage en python vous savez comment utiliser Pydantic.
- Aide votre IDE/linter/cerveau:
 - Parce que les structures de données de pydantic consistent seulement en une instance de classe que vous définissez; l'auto-complétion, le linting, mypy et votre intuition devrait être largement suffisante pour valider vos données.

- **Rapide:**
 - Dans les [benchmarks](#) Pydantic est plus rapide que toutes les autres librairies testées.
- Valide les **structures complexes**:
 - Utilise les modèles hiérarchique de Pydantic, le `type` Python pour les `Lists`, `Dict`, etc.
 - Et les validateurs permettent aux schémas de données complexes d'être clairement et facilement définis, validés et documentés sous forme d'un schéma JSON.
 - Vous pouvez avoir des objets **JSON fortement imbriqués** tout en ayant, pour chacun, de la validation et des annotations.
- **Renouvelable:**
 - Pydantic permet de définir de nouveaux types de données ou vous pouvez étendre la validation avec des méthodes sur un modèle décoré avec le décorateur de validation
- 100% de couverture de test.