

Recursos

Recursos do FastAPI

FastAPI te oferece o seguinte:

Baseado em padrões abertos

- OpenAPI para criação de APIs, incluindo declarações de operações de caminho, parâmetros, requisições de corpo, segurança etc.
- Modelo de documentação automática com JSON Schema (já que o OpenAPI em si é baseado no JSON Schema).
- Projetado em cima desses padrões após um estudo metódico, em vez de uma reflexão breve.
- Isso também permite o uso de **geração de código do cliente** automaticamente em muitas linguagens.

Documentação automática

Documentação interativa da API e navegação *web* da interface de usuário. Como o *framework* é baseado no OpenAPI, há várias opções, 2 incluídas por padrão.

- Swagger UI, com navegação interativa, chame e teste sua API diretamente do navegador.
- Documentação alternativa da API com ReDoc.

Apenas Python moderno

Tudo é baseado no padrão das declarações de **tipos do Python 3.6** (graças ao Pydantic). Nenhuma sintaxe nova para aprender. Apenas o padrão moderno do Python.

Se você precisa refrescar a memória rapidamente sobre como usar tipos do Python (mesmo que você não use o FastAPI), confira esse rápido tutorial: Tipos do Python.

Você escreve Python padrão com tipos:

```
from datetime import date

from pydantic import BaseModel

# Declare uma variável como str
# e obtenha suporte do editor dentro da função
def main(user_id: str):
    return user_id
```

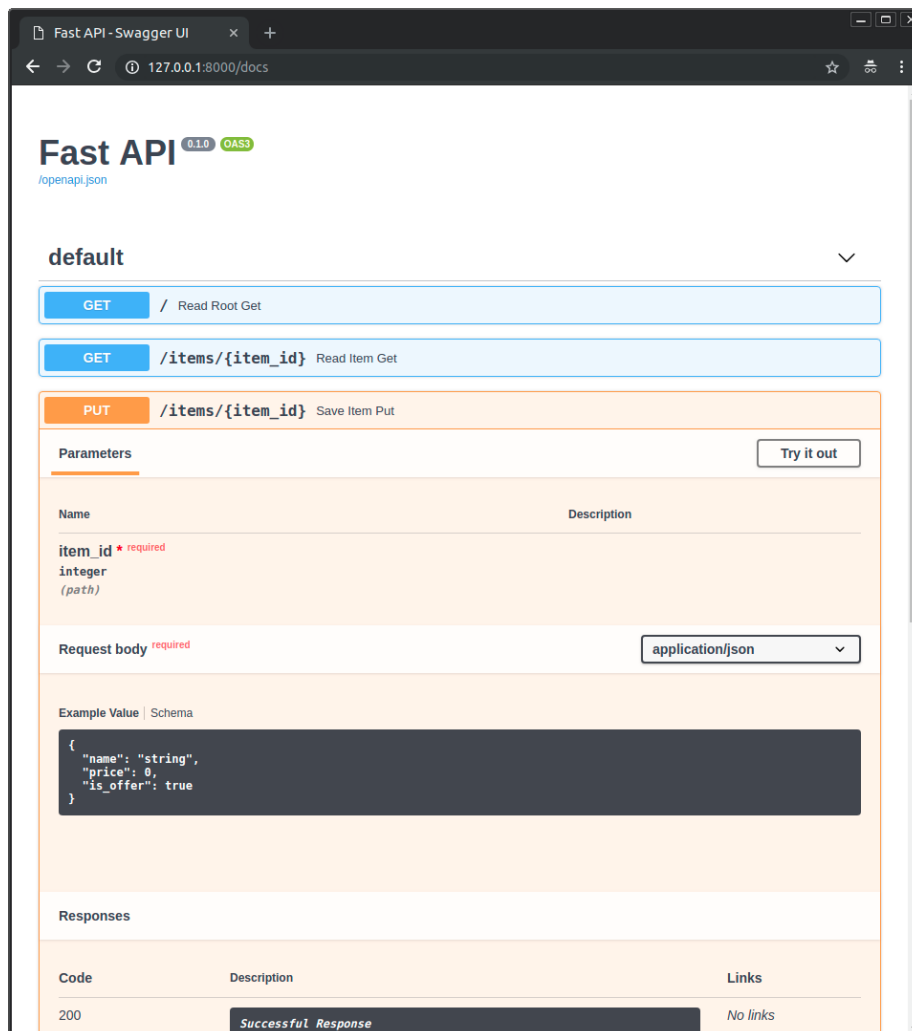


Figure 1: Interação Swagger UI

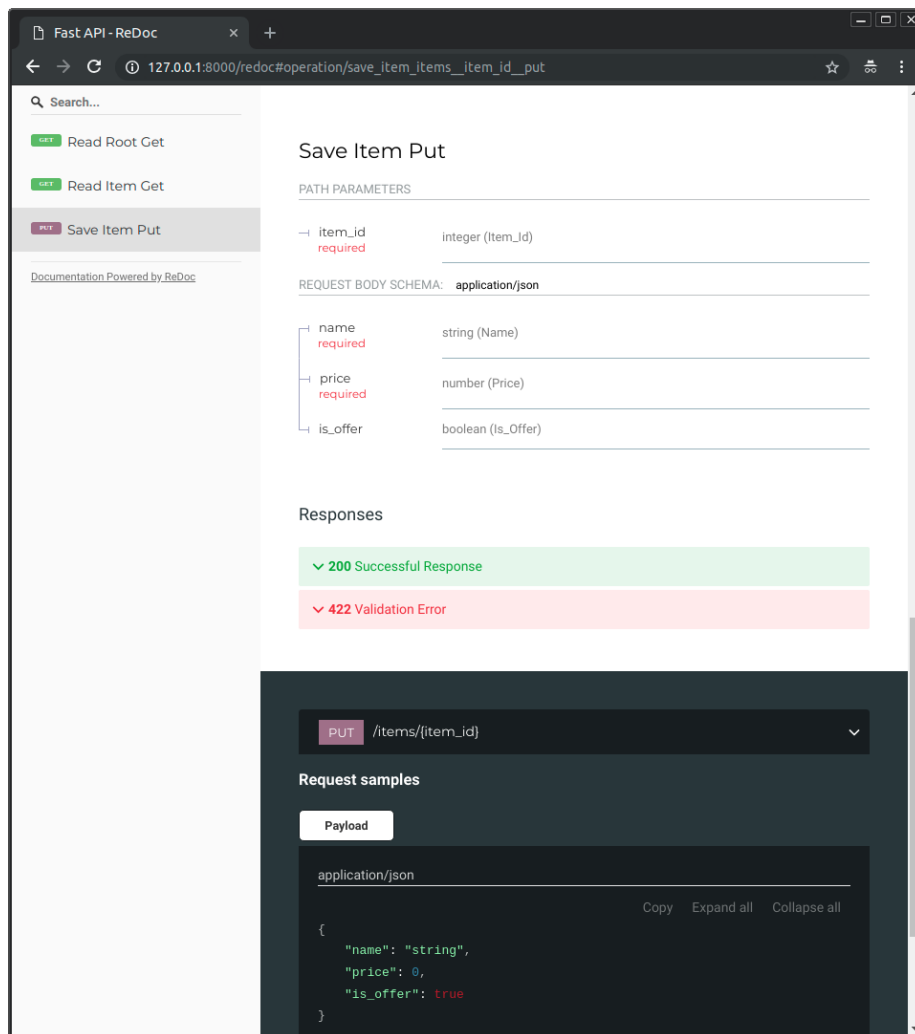


Figure 2: ReDoc

```
# Um modelo do Pydantic
```

```
class User(BaseModel):  
    id: int  
    name: str  
    joined: date
```

Que então pode ser usado como:

```
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` quer dizer:

Passe as chaves e valores do dicionário `second_user_data` diretamente como argumentos chave

Suporte de editores

Todo o *framework* foi projetado para ser fácil e intuitivo de usar, todas as decisões foram testadas em vários editores antes do início do desenvolvimento, para garantir a melhor experiência de desenvolvimento.

Na última pesquisa do desenvolvedor Python ficou claro que o recurso mais utilizado é o “auto completar”.

Todo o *framework* **FastAPI** é feito para satisfazer isso. Auto completção funciona em todos os lugares.

Você raramente precisará voltar à documentação.

Aqui está como o editor poderá te ajudar:

- no Visual Studio Code:
- no PyCharm:

Você terá completção do seu código que você poderia considerar impossível antes. Como por exemplo, a chave `price` dentro do corpo JSON (que poderia ter sido aninhado) que vem de uma requisição.

Sem a necessidade de digitar nomes de chaves erroneamente, ir e voltar entre documentações, ou rolar pela página para descobrir se você utilizou `username` or `user_name`.

```
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.pr, "item_id": item_id}
26
```

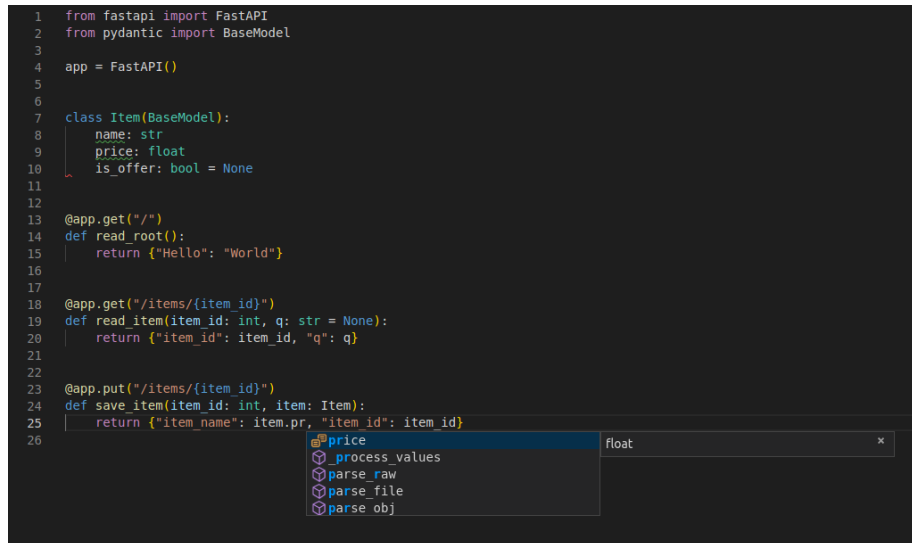


Figure 3: editor support

```
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.pr, "item_id": item_id}
26
```

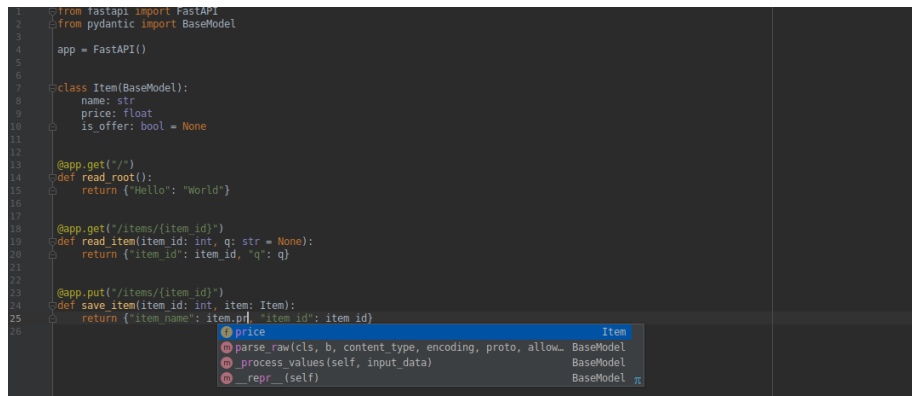


Figure 4: editor support

Breve

Há **padrões** sensíveis para tudo, com configurações adicionais em todos os lugares. Todos os parâmetros podem ser regulados para fazer o que você precisa e para definir a API que você necessita.

Por padrão, tudo “**simplesmente funciona**”.

Validação

- Validação para a maioria dos (ou todos?) **tipos de dados** do Python, incluindo:
 - objetos JSON (`dict`).
 - arrays JSON (`list`), definindo tipos dos itens.
 - campos String (`str`), definindo tamanho mínimo e máximo.
 - Numbers (`int`, `float`) com valores mínimos e máximos, etc.
- Validação de tipos mais exóticos, como:
 - URL.
 - Email.
 - UUID.
 - ... e outros.

Toda a validação é controlada pelo robusto e bem estabelecido **Pydantic**.

Segurança e autenticação

Segurança e autenticação integradas. Sem nenhum compromisso com bancos de dados ou modelos de dados.

Todos os esquemas de seguranças definidos no OpenAPI, incluindo:

- HTTP Basic.
- **OAuth2** (também com **tokens JWT**). Confira o tutorial em OAuth2 com JWT.
- Chaves de API em:
 - Headers.
 - parâmetros da Query.
 - Cookies etc.

Além disso, todos os recursos de seguranças do Starlette (incluindo **cookies de sessão**).

Tudo construído como ferramentas e componentes reutilizáveis que são fáceis de integrar com seus sistemas, armazenamento de dados, banco de dados relacionais e não-relacionais etc.

Injeção de dependência

FastAPI inclui um sistema de injeção de dependência extremamente fácil de usar, mas extremamente poderoso.

- Mesmo dependências podem ter dependências, criando uma hierarquia ou “**grafo**” de dependências.
- Tudo **automaticamente controlado** pelo *framework*.
- Todas as dependências podem pedir dados das requisições e **ampliar** as restrições e documentação automática da **operação de caminho**.
- **Validação automática** mesmo para parâmetros da *operação de caminho* definidos em dependências.
- Suporte para sistemas de autenticação complexos, **conexões com banco de dados** etc.
- **Sem comprometer** os bancos de dados, *frontends* etc. Mas fácil integração com todos eles.

“Plug-ins” ilimitados

Ou, de outra forma, sem a necessidade deles, importe e use o código que precisar.

Qualquer integração é projetada para ser tão simples de usar (com dependências) que você pode criar um “plug-in” para suas aplicações com 2 linhas de código usando a mesma estrutura e sintaxe para as suas *operações de caminho*.

Testado

- 100% de cobertura de testes.
- 100% do código utiliza type annotations.
- Usado para aplicações em produção.

Recursos do Starlette

FastAPI é totalmente compatível com (e baseado no) Starlette. Então, qualquer código adicional Starlette que você tiver, também funcionará.

FastAPI é na verdade uma sub-classe do **Starlette**. Então, se você já conhece ou usa Starlette, a maioria das funcionalidades se comportará da mesma forma.

Com **FastAPI**, você terá todos os recursos do **Starlette** (já que FastAPI é apenas um Starlette com esteróides):

- Desempenho realmente impressionante. É um dos *frameworks* Python disponíveis mais rápidos, a par com o **NodeJS** e **Go**.
- Suporte a **WebSocket**.
- Suporte a **GraphQL**.
- Tarefas em processo *background*.
- Eventos na inicialização e encerramento.
- Cliente de testes construído sobre **requests**.
- Respostas em **CORS**, GZip, Static Files, Streaming.
- Suporte a **Session** e **Cookie**.
- 100% de cobertura de testes.
- 100% do código utilizando *type annotations*.

Recursos do Pydantic

FastAPI é totalmente compatível com (e baseado no) Pydantic. Então, qualquer código Pydantic adicional que você tiver, também funcionará.

Incluindo bibliotecas externas também baseadas no Pydantic, como ORMs e ODMs para bancos de dados.

Isso também significa que em muitos casos você poderá passar o mesmo objeto que você receber de uma requisição **diretamente para o banco de dados**, já que tudo é validado automaticamente.

O mesmo se aplica no sentido inverso, em muitos casos você poderá simplesmente passar o objeto que você recebeu do banco de dados **diretamente para o cliente**.

Com **FastAPI** você terá todos os recursos do **Pydantic** (já que FastAPI utiliza o Pydantic para todo o controle dos dados):

- **Sem pegadinhas:**
 - Sem novas definições de esquema de micro-linguagem para aprender.
 - Se você conhece os tipos do Python, você sabe como usar o Pydantic.
- Vai bem com o/a seu/sua **IDE/linter/cérebro**:
 - Como as estruturas de dados do Pydantic são apenas instâncias de classes que você define, a auto completção, *linting*, *mypy* e a sua intuição devem funcionar corretamente com seus dados validados.
- **Rápido:**
 - em *benchmarks*, o Pydantic é mais rápido que todas as outras bibliotecas testadas.
- Valida **estruturas complexas**:
 - Use modelos hierárquicos do Pydantic, `List` e `Dict` do `typing` do Python, etc.
 - Validadores permitem que esquemas de dados complexos sejam limpos e facilmente definidos, conferidos e documentados como JSON Schema.
 - Você pode ter **JSONs aninhados** profundamente e tê-los todos validados e anotados.
- **Extensível:**
 - Pydantic permite que tipos de dados personalizados sejam definidos ou você pode estender a validação com métodos em um modelo decorado com seu decorador de validador.
- 100% de cobertura de testes.