

# 機能

## FastAPIの機能

FastAPI は以下の機能をもちます:

### オープンスタンダード準拠

- API作成のための[OpenAPI](#)。これは、`path operations`の宣言、パラメータ、ボディリクエスト、セキュリティなどを含んでいます。
- [JSONスキーマ](#)を使用したデータモデルのドキュメント自動生成（OpenAPIはJSONスキーマに基づいている）。
- 綿密な調査の結果、上層に後付けするのではなく、これらの基準に基づいて設計されました。
- これにより、多くの言語で自動 **クライアントコード生成** が可能です。

### 自動ドキュメント生成

対話的なAPIドキュメントと探索的なwebユーザーインターフェースを提供します。フレームワークはOpenAPIを基にしているため、いくつかのオプションがあり、デフォルトで2つ含まれています。

- [Swagger UI](#)で、インタラクティブな探索をしながら、ブラウザから直接APIを呼び出してテストが行えます。

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

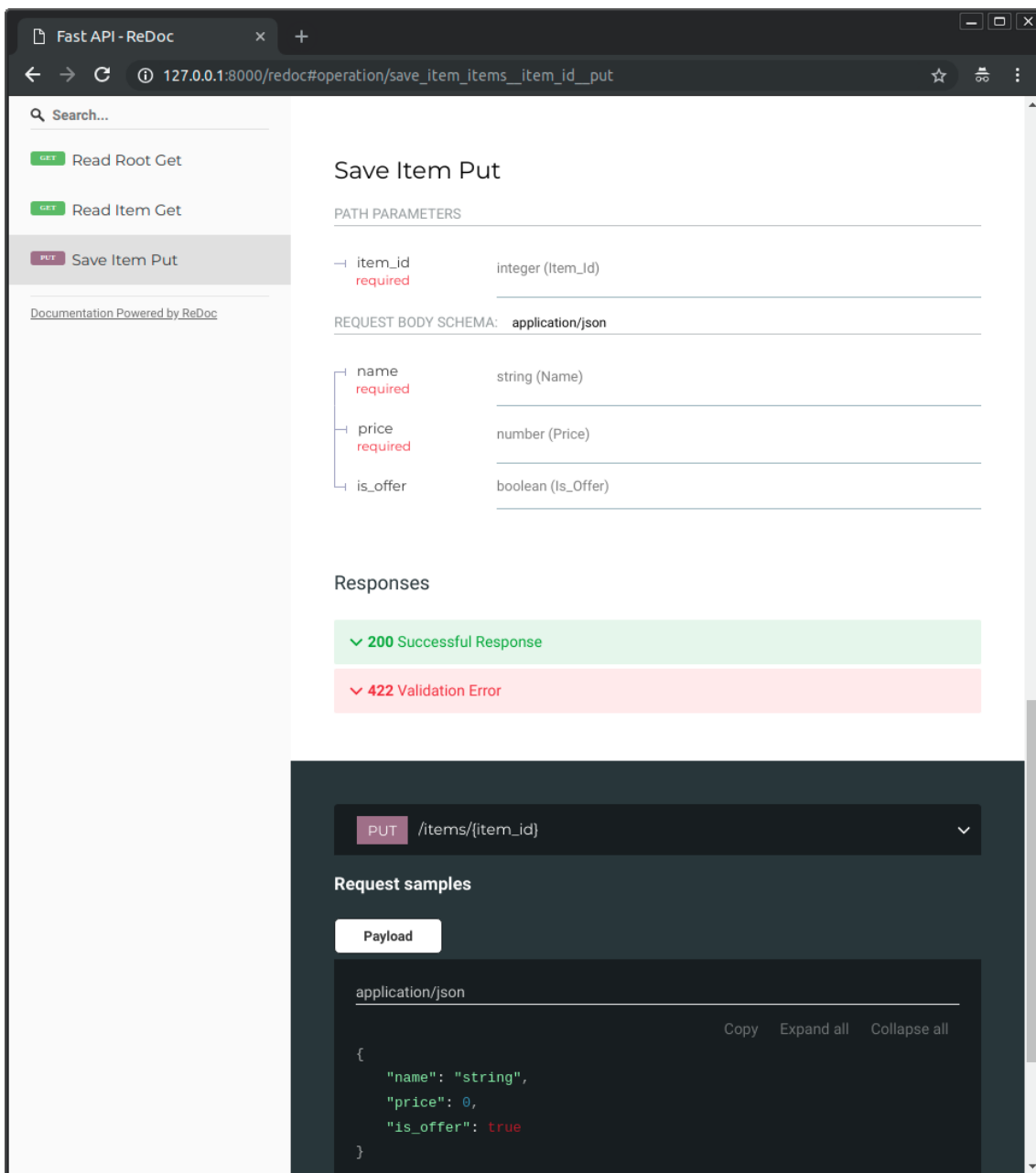
Example Value | Schema

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

Responses

Code	Description	Links
200	Successful Response	No links

- [ReDoc](#)を使用したもう一つのAPIドキュメント生成。



## 現代的なPython

FastAPIの機能はすべて、標準のPython 3.6型宣言に基づいています（Pydanticの功績）。新しい構文はありません。ただの現代的な標準のPythonです。

（FastAPIを使用しない場合でも）Pythonの型の使用方法について簡単な復習が必要な場合は、短いチュートリアル（[Python Types](#), internal-link target=\_blank）を参照してください。

型を使用した標準的なPythonを記述します：

```
from datetime import date

from pydantic import BaseModel
```

```
# Declare a variable as a str
# and get editor support inside the function
def main(user_id: str):
    return user_id

# A Pydantic model
class User(BaseModel):
    id: int
    name: str
    joined: date
```

これは以下のように用いられます:

```
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 "情報" `**second_user_data` は以下を意味します:

``second_user_data`` 辞書のキーと値を直接、キーと値の引数として渡します。これは、``User(id=4, name="Mary", joined="2018-11-30")`` と同等です。

## エディタのサポート

すべてのフレームワークは使いやすく直感的に使用できるように設計されており、すべての決定は開発を開始する前でも複数のエディターでテストされ、最高の開発体験が保証されます。

前回のPython開発者調査では、[最も使用されている機能が「オートコンプリート」であることが明らかになりました。](#)

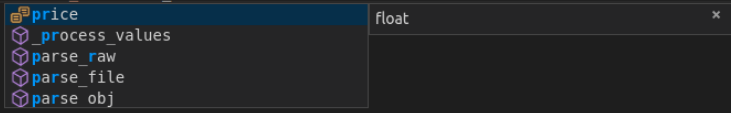
**FastAPI** フレームワークは、この要求を満たすことを基本としています。オートコンプリートはどこでも機能します。

ドキュメントに戻る必要はほとんどありません。

エディターがどのように役立つかを以下に示します:

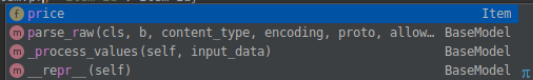
- [Visual Studio Code](#) の場合:

```
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
```



- [PyCharm](#)の場合:

```
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
```



以前は不可能だと思っていたコードでさえ補完されます。例えば、リクエストからのJSONボディ（ネストされている可能性がある）内の `price` キーです。

間違ったキー名を入力したり、ドキュメント間を行き来したり、上下にスクロールして `username` と `user_name` のどちらを使用したか調べたりする必要はもうありません。

## 簡潔

すべてに適切なデフォルトがあり、オプションの構成ができます。必要なことを実行し、必要なAPIを定義するためにすべてのパラメーターを調整できます。

ただし、デフォルトでもすべて うまくいきます。

## 検証

- 以下の様な、ほとんどの（すべての？）Python **データ型**の検証:
  - JSONオブジェクト（`dict`）
  - 項目の型を定義するJSON配列（`list`）
  - 最小長と最大長のある文字列（`str`）フィールド
  - 最小値と最大値のある数値（`int`、`float`）
- よりエキゾチックな型の検証:
  - URL
  - Eメール
  - UUID
  - ...その他

すべての検証は、確立された堅牢な **Pydantic** によって処理されます。

## セキュリティと認証

セキュリティと認証が統合されています。データベースまたはデータモデルについても妥協していません。

以下のOpenAPIで定義されているすべてのセキュリティスキームを含む:

- HTTPベーシック
- **OAuth2（JWTトークンも使用）**。JWTを使用したOAuth2のチュートリアル（[OAuth2 with JWT](#){internal-link target=\_blank}）を確認してください。
- APIキー：
  - ヘッダー
  - クエリパラメータ
  - クッキー、等

さらに、Starletteのすべてのセキュリティ機能も含まれます（**セッションCookie**を含む）。

これらは、システム、データストア、リレーショナルデータベース、NoSQLデータベースなどと簡単に統合できる再利用可能なツールとコンポーネントとして構築されています。

## 依存性の注入（Dependency Injection）

FastAPIには非常に使いやすく、非常に強力な**依存性の注入**システムを備えています。

- 依存関係でさえも依存関係を持つことができ、階層または**依存関係の"グラフ"**を作成することができます。
- フレームワークによってすべて**自動的に処理**されます。
- すべての依存関係はリクエストからのデータを要請できて、**path operationsの制約と自動ドキュメンテーションを拡張**できます。
- 依存関係で定義された *path operation* パラメータも**自動検証**が可能です。
- 複雑なユーザー認証システム、**データベース接続**などのサポート
- データベース、フロントエンドなど**に対する妥協はありません**。それらすべてと簡単に統合できます。

## 無制限の「プラグイン」

他の方法では、それらを必要とせず、必要なコードをインポートして使用します。

統合は非常に簡単に使用できるように設計されており（依存関係を用いて）、*path operations* で使用されているのと同じ構造と構文を使用して、2行のコードでアプリケーションの「プラグイン」を作成できます。

## テスト

- テストカバレッジ 100%
- 型アノテーション 100%のコードベース
- 本番アプリケーションで使用されます

## Starletteの機能

FastAPIは、[Starlette](#)と完全に互換性があります（そしてベースになっています）。したがって、追加のStarletteコードがあれば、それも機能します。

FastAPI は実際には Starlette のサブクラスです。したがって、Starletteをすでに知っているか使用している場合は、ほとんどの機能が同じように機能します。

FastAPIを使用すると、以下のような、**Starlette**のすべての機能を利用できます（FastAPIはStarletteを強化したものにすぎないため）：

- 見事なパフォーマンス。 [NodeJSおよびGoに匹敵する、最速のPythonフレームワークの1つです。](#)
- **WebSocket**のサポート
- **GraphQL**のサポート
- プロセス内バックグラウンドタスク
- 起動およびシャットダウンイベント
- `requests` に基づいて構築されたテストクライアント
- **CORS**、GZip、静的ファイル、ストリーミング応答
- **セッションとCookie**のサポート
- テストカバレッジ100%
- 型アノテーション100%のコードベース

## Pydanticの特徴

FastAPIは[Pydantic](#)と完全に互換性があります（そしてベースになっています）。したがって、追加のPydanticコードがあれば、それも機能します。

データベースのためにORMsや、ODMsなどの、Pydanticに基づく外部ライブラリを備えています。

これは、すべてが自動的に検証されるため、多くの場合、リクエストから取得したオブジェクトをデータベースに直接渡すことができるということを意味しています。

同じことがその逆にも当てはまり、多くの場合、データベースから取得したオブジェクトをクライアントに直接渡すことができます。

FastAPIを使用すると、**Pydantic**のすべての機能を利用できます（FastAPIがPydanticに基づいてすべてのデータ処理を行っているため）。

- **brainfuckなし:**
  - スキーマ定義のためのマイクロ言語を新たに学習する必要はありません。
  - Pythonの型を知っている場合は、既にPydanticの使用方法を知っているに等しいです。
- ユーザーの IDE/リンター/思考 とうまく連携します:
  - Pydanticのデータ構造は、ユーザーが定義するクラスの単なるインスタンスであるため、オートコンプリート、リンティング、mypy、およびユーザーの直感はすべて、検証済みのデータで適切に機能するはずです。
- **高速:**
  - ベンチマーク では、Pydanticは他のすべてのテスト済みライブラリよりも高速です。
- **複雑な構造を検証:**
  - 階層的なPydanticモデルや、Pythonの「`typing`」の「`list`」と「`dict`」などの利用。
  - バリデーターにより、複雑なデータスキーマを明確かつ簡単に定義、チェックし、JSONスキーマとして文書化できます。
  - 深くネストされたJSONオブジェクトを作成し、それらすべてを検証してアノテーションを付けることができます。
- **拡張可能:**
  - Pydanticでは、カスタムデータ型を定義できます。または、バリデーターデコレーターで装飾されたモデルのメソッドを使用して検証を拡張できます。
- テストカバレッジ 100%。