

Características

Características de FastAPI

FastAPI te provee lo siguiente:

Basado en estándares abiertos

- [OpenAPI](#) para la creación de APIs, incluyendo declaraciones de `path operations`, parámetros, `body`, requests, seguridad, etc.
- Documentación automática del modelo de datos con [JSON Schema](#) (dado que OpenAPI mismo está basado en JSON Schema).
- Diseñado alrededor de estos estándares después de un estudio meticuloso. En vez de ser una capa añadida a último momento.
- Esto también permite la **generación automática de código de cliente** para muchos lenguajes.

Documentación automática

Documentación interactiva de la API e interfaces web de exploración. Hay múltiples opciones, dos incluidas por defecto, porque el framework está basado en OpenAPI.

- [Swagger UI](#), con exploración interactiva, llama y prueba tu API directamente desde tu navegador.

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

- Documentación alternativa de la API con [ReDoc](#).

Fast API - ReDoc

127.0.0.1:8000/redoc#operation/save_item_items__item_id__put

Search...

GET Read Root Get

GET Read Item Get

PUT Save Item Put

Documentation Powered by ReDoc

Save Item Put

PATH PARAMETERS

item_id	integer (Item_Id)
required	

REQUEST BODY SCHEMA: application/json

name	string (Name)
required	
price	number (Price)
required	
is_offer	boolean (Is_Offer)

Responses

- ✓ 200 Successful Response
- ✓ 422 Validation Error

PUT /items/{item_id}

Request samples

Payload

application/json

Copy Expand all Collapse all

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

Simplemente Python moderno

Todo está basado en las declaraciones de tipo de **Python 3.6** estándar (gracias a Pydantic). No necesitas aprender una sintaxis nueva, solo Python moderno.

Si necesitas un repaso de 2 minutos de cómo usar los tipos de Python (así no uses FastAPI) prueba el tutorial corto: [Python Types](#) (internal-link target=_blank).

Escribes Python estándar con tipos así:

```
from datetime import date

from pydantic import BaseModel
```

```
# Declaras la variable como un str
# y obtienes soporte del editor dentro de la función
def main(user_id: str):
    return user_id

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

Este puede 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` significa:

Pasa las `<abbr title="en español key se refiere a la guía de un diccionario">keys</abbr>` y los valores del dict ``second_user_data`` directamente como argumentos de key-value, equivalente a: ``User(id=4, name="Mary", joined="2018-11-30")``

Soporte del editor

El framework fue diseñado en su totalidad para ser fácil e intuitivo de usar. Todas las decisiones fueron probadas en múltiples editores antes de comenzar el desarrollo para asegurar la mejor experiencia de desarrollo.

En la última encuesta a desarrolladores de Python fue claro que [la característica más usada es el "autocompletado"](#).

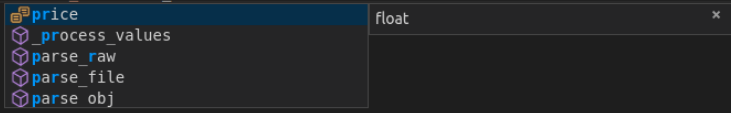
El framework **FastAPI** está creado para satisfacer eso. El autocompletado funciona en todas partes.

No vas a tener que volver a la documentación seguido.

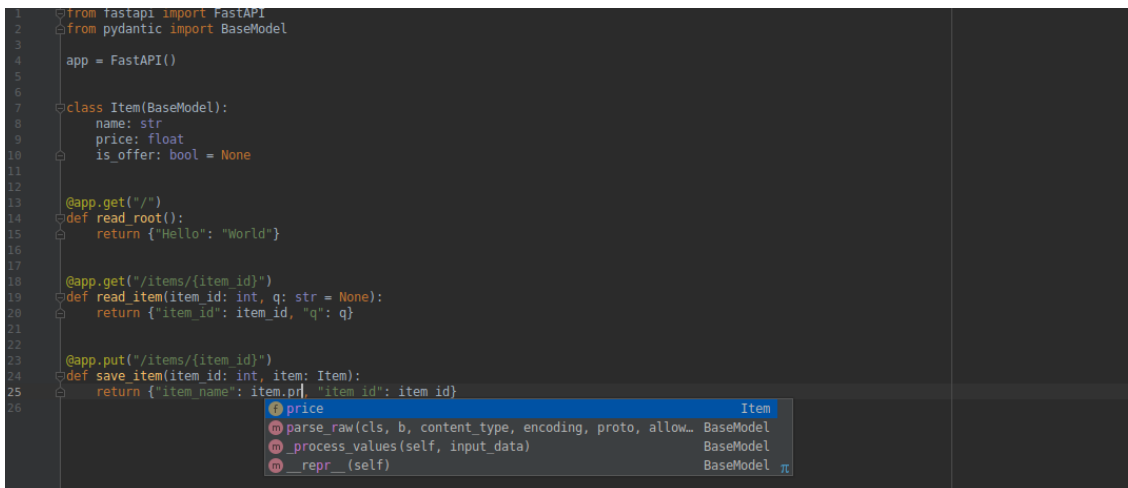
Así es como tu editor te puede ayudar:

- en [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
```



- en [PyCharm](#):



Obtendrás completado para tu código que podrías haber considerado imposible antes. Por ejemplo, el key `price` dentro del JSON body (que podría haber estado anidado) que viene de un request.

Ya no pasará que escribas los nombres de key equivocados, o que tengas que revisar constantemente la documentación o desplazarte arriba y abajo para saber si usaste `username` o `user_name`.

Corto

Tiene **configuraciones por defecto** razonables para todo, con configuraciones opcionales en todas partes. Todos los parámetros pueden ser ajustados para tus necesidades y las de tu API.

Pero, todo **simplemente funciona** por defecto.

Validación

- Validación para la mayoría (¿o todos?) los **tipos de datos** de Python incluyendo:
 - Objetos JSON (`dict`).
 - JSON array (`list`) definiendo tipos de ítem.
 - Campos de texto (`str`) definiendo longitudes mínimas y máximas.
 - Números (`int` , `float`) con valores mínimos y máximos, etc.
- Validación para tipos más exóticos como:
 - URL.
 - Email.
 - UUID.
 - ...y otros.

Toda la validación es manejada por **Pydantic**, que es robusto y sólidamente establecido.

Seguridad y autenticación

La seguridad y la autenticación están integradas. Sin ningún compromiso con bases de datos ni modelos de datos.

Todos los schemes de seguridad están definidos en OpenAPI incluyendo:

- HTTP Basic.
- **OAuth2** (también con **JWT tokens**). Prueba el tutorial en [OAuth2 with JWT](#) {internal-link target=_blank}.
- API keys en:
 - Headers.
 - Parámetros de Query.
 - Cookies, etc.

Más todas las características de seguridad de Starlette (incluyendo **session cookies**).

Todo ha sido construido como herramientas y componentes reutilizables que son fácilmente integrados con tus sistemas, almacenamiento de datos, bases de datos relacionales y no relacionales, etc.

Dependency Injection

FastAPI incluye un sistema de **Dependency Injection** extremadamente poderoso y fácil de usar.

- Inclusive las dependencias pueden tener dependencias creando una jerarquía o un **"grafo" de dependencias**.
- Todas son **manejadas automáticamente** por el framework.
- Todas las dependencias pueden requerir datos de los requests y aumentar las restricciones del *path operation* y la documentación automática.
- **Validación automática** inclusive para parámetros del *path operation* definidos en las dependencias.
- Soporte para sistemas complejos de autenticación de usuarios, **conexiones con bases de datos**, etc.
- **Sin comprometerse** con bases de datos, frontends, etc. Pero permitiendo integración fácil con todos ellos.

"Plug-ins" ilimitados

O dicho de otra manera, no hay necesidad para "plug-ins". Importa y usa el código que necesites.

Cualquier integración está diseñada para que sea tan sencilla de usar (con dependencias) que puedas crear un "plug-in" para tu aplicación en dos líneas de código usando la misma estructura y sintaxis que usaste para tus *path operations*.

Probado

- Cobertura de pruebas al 100%.
- Base de código 100% anotada con tipos.
- Usado en aplicaciones en producción.

Características de Starlette

FastAPI está basado y es completamente compatible con [Starlette](#). Tanto así, que cualquier código de Starlette que tengas también funcionará.

`FastAPI` es realmente una sub-clase de `Starlette`. Así que, si ya conoces o usas Starlette, muchas de las características funcionarán de la misma manera.

Con **FastAPI** obtienes todas las características de **Starlette** (porque FastAPI es simplemente Starlette en esteroides):

- Desempeño realmente impresionante. Es uno [de los frameworks de Python más rápidos, a la par con NodeJS y Go](#).
- Soporte para **WebSocket**.
- Soporte para **GraphQL**.
- Tareas en background.
- Eventos de startup y shutdown.
- Cliente de pruebas construido con `requests`.
- **CORS**, GZip, Static Files, Streaming responses.
- Soporte para **Session and Cookie**.
- Cobertura de pruebas al 100%.
- Base de código 100% anotada con tipos.

Características de Pydantic

FastAPI está basado y es completamente compatible con [Pydantic](#). Tanto así, que cualquier código de Pydantic que tengas también funcionará.

Esto incluye a librerías externas basadas en Pydantic como ORMs y ODMs para bases de datos.

Esto también significa que en muchos casos puedes pasar el mismo objeto que obtuviste de un request **directamente a la base de datos**, dado que todo es validado automáticamente.

Lo mismo aplica para el sentido contrario. En muchos casos puedes pasarle el objeto que obtienes de la base de datos **directamente al cliente**.

Con **FastAPI** obtienes todas las características de **Pydantic** (dado que FastAPI está basado en Pydantic para todo el manejo de datos):

- **Sin dificultades para entender:**
 - No necesitas aprender un nuevo micro-lenguaje de definición de schemas.
 - Si sabes tipos de Python, sabes cómo usar Pydantic.
- Interactúa bien con tu IDE/linter/cerebro:
 - Porque las estructuras de datos de Pydantic son solo instancias de clases que tu defines, el auto-completado, el linting, mypy y tu intuición deberían funcionar bien con tus datos validados.
- **Rápido:**
 - En [benchmarks](#) Pydantic es más rápido que todas las otras libraries probadas.
- Valida **estructuras complejas**:

- Usa modelos jerárquicos de modelos de Pydantic, `typing` de Python, `List` y `Dict`, etc.
- Los validadores también permiten que se definan fácil y claramente schemas complejos de datos. Estos son chequeados y documentados como JSON Schema.
- Puedes tener objetos de **JSON profundamente anidados** y que todos sean validados y anotados.
- **Extensible:**
 - Pydantic permite que se definan tipos de datos a la medida o puedes extender la validación con métodos en un modelo decorado con el decorador de validación.
- Cobertura de pruebas al 100%.