

# Introducción a Fast API

Hilario Tun



Es un micro framework para crear APIs con python 3.7 o superior

- Alto rendimiento
- Fácil de aprender
- Fácil de programar
- Listo para producción



# Porque?

Usado por la industria

- Microsoft, Uber, Netflix, etc.
- Buena aceptación en github

Desempeño

- Entre los frameworks con mejor desempeño alcanzado con python

---

# Otras características

Inyección de dependencias

WebSockets

Archivos

Tareas en segundo plano

Integración con GraphQL o cualquier ORM

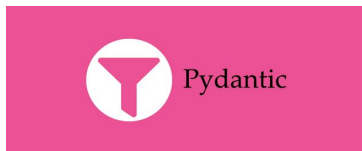
Soporte para cualquier editor



¿Cómo funciona?

# Base

## Pydantic



Validación y administración de preferencias utilizando los type annotations de Python.

## Starlette



Framework/Toolkit ligero ideal para construir servicios web asíncronos en Python.

## Uvicorn



Una implementación de servidor web ASGI para Python.

# Curva de aprendizaje corta

Sin esfuerzo extra



¿Cómo lo hace?



# Basado en estándares

## Python

- f-strings
- Type hints

## Uso de

1. OpenAPI
2. JSON Schema
3. OAuth2
4. API docs automáticos

---

# Type hints?

Introducido en Python 3.5+

Utilizado por herramientas de terceros

IDE's

Linters

# Type hints?

Autocompletado para todo

```
1 def get_full_name(first_name: str, last_name: str):
```

```
2 |     full_name = first_name.
```

You, a few seconds ago • Uncommitted changes

★ format

★ join

★ split

★ encode

capitalize

casefold

center

count

endswith

expandtabs

find

format map

`str.format(self, *args, **kwargs)` \*

`S.format(args, *kwargs) -> str`

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

# Type hints?

## Revisión de tipos y errores

```
1 def get_name_with_age(name: str, age: int):  
2  
3     [mypy] Unsupported operand types for + ("str" and "int")  
4     [error]  
5     name_with_age = name + " is this old: " + age  
6     return name_with_age  
7
```

# Type hints?

## Tipos anidados

```
1 from typing import List
2
3 def process_items(items: List[str]):
4     for item in items:
5         print([item.])
6
```

- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format
- format\_map
- index
- isalnum

str.capitalize(self)

S.capitalize() -> str

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

# Type hints?

## Classes como tipos

```
1 class Person:
2     def __init__(self, name: str):
3         self.name = name
4
5
6 def get_person_name(one_person: Person):
7     return one_person.name
8
```



A screenshot of an IDE's autocomplete menu. The menu is open for the variable 'name' in the function 'get\_person\_name'. It lists various attributes and methods of the 'Person' class, including 'next', 'bases', 'class', 'delattr', 'dir', 'doc', 'eq', 'format', 'ge', 'getattribute', and 'gt'. The 'name' attribute is highlighted at the top of the list.

- ★ name
- next
- bases
- class
- delattr
- dir
- doc
- eq
- format
- ge
- getattribute
- gt

str

# Type hints?

## Classes como tipos

```
1 class Person:
2     def __init__(self, name: str):
3         self.name = name
4
5
6 def get_person_name(one_person: Person):
7     return one_person.name
8
```



A screenshot of an IDE's autocomplete menu. The menu is open for the variable 'name' in the function 'get\_person\_name'. It lists various attributes and methods of the 'Person' class, including 'next', 'bases', 'class', 'delattr', 'dir', 'doc', 'eq', 'format', 'ge', 'getattr', and 'gt'. The 'name' attribute is highlighted at the top of the list.

- ★ name
- next
- bases
- class
- delattr
- dir
- doc
- eq
- format
- ge
- getattr
- gt

str

# Pydantic

```
class Person:
    def __init__(self, name: str):
        self.name = name

def get_person_name(one_person: Person):
    return one_person.name
```

1. Validará los valores
  2. Los convertirá al tipo apropiado (si ese es el caso)
  3. Te dará un objeto con todos los datos
-



# Aplicación básica

```
from fastapi import Depends, FastAPI,

app = FastAPI()

@app.get("/weather/{place}/", tags=["Basic usage"])
def weather(place: str, rain: bool = False):
    if rain:
        return {"Hello": f"Hello {place}, today is a rainy day"}
    return {"Hello": f"Hello {place}, today is a sunny day"}
```

# Documentación Automática

GET

/weather/{place}/ Weather

^

Parameters

Try it out

Name	Description
<b>place</b> * required string (path)	<input type="text" value="place"/>
rain boolean (query)	Default value : false <input type="text" value="false"/>

Responses

Code	Description	Links
200	Successful Response  Media type <input type="text" value="application/json"/> Controls Accept header. <b>Example Value</b>   Schema	No links

# Documentación Personalizada

```
@app.get("/custom", tags=["Docs"])
def custom_docs(days: int = Query(default=1, gt=0, le=10, description="Number of
days")):
    """
    ### Return the following info:

    - **message**: a message to say hello world

    ### This allow md
    - *Line 1*: Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    - **Line 2**: sed do eiusmod tempor incididunt ut labore et dolore
    - *Line 3*: magna aliqua. Ut enim ad minim veniam.
    """
    return {"message": "Hello World"}
```

# Bases de datos

Compatibilidad con múltiples ORMs

The SQLAlchemy logo, featuring the word "SQL" in a black, stylized font and "Alchemy" in a red, cursive font.The SQLModel logo, featuring a purple database cylinder icon followed by the text "SQLModel" in a purple, sans-serif font.

*SQL databases in Python,  
designed for simplicity, compatibility, and robustness.*

The django logo, featuring the word "django" in a white, lowercase, sans-serif font on a dark green rectangular background.The mongoDB logo, featuring a green leaf icon followed by the text "mongoDB" in a grey, lowercase, sans-serif font, with a registered trademark symbol.

# Async

Se pueden declarar funciones asíncronas

```
@app.get("/waiting/", tags=['Async'])
async def get_resource():
    async with httpx.AsyncClient() as client:
        response = await client.get(f"https://httpstat.us/201?sleep=5000",
        timeout=None)
        print("Done")
        return {'status_code': response.status_code}
```

# Async

Se pueden declarar funciones asíncronas

```
@app.get("/sleep", tags=['Async'])
async def sleep_response(ms: int):
    urls = [
        f"https://httpstat.us/200?sleep= {ms}",
        f"https://httpstat.us/200?sleep= {ms}",
        f"https://httpstat.us/200?sleep= {ms}"
    ]
    tasks = [make_request(url) for url in urls]
    responses = await asyncio.gather(*tasks)
    for response in responses:
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code, detail=response.text)
    return {'status_code': response.status_code}
```

# WebSockets

Con soporte para websockets

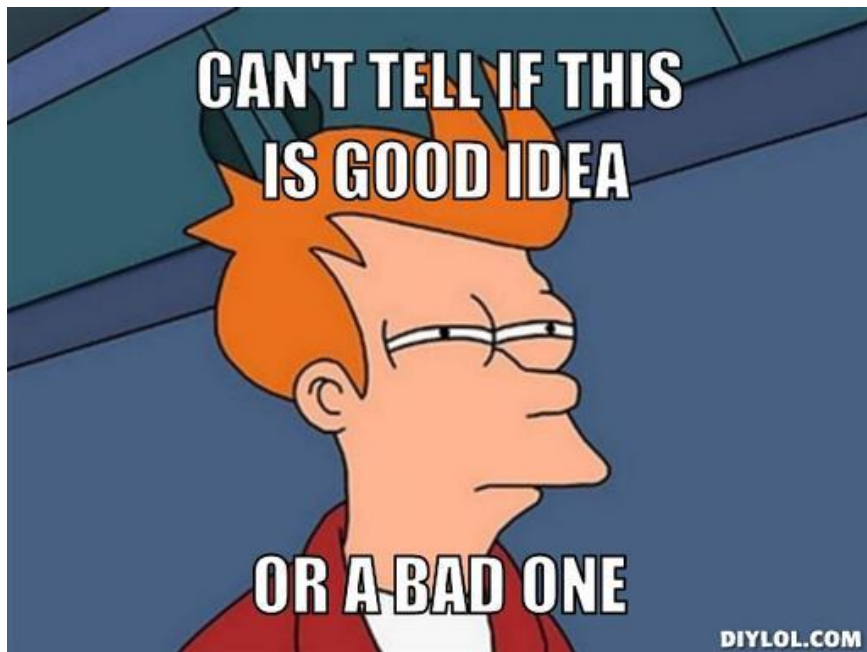
```
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")
```

# Tareas en segundo plano

Se ejecutan después de la respuesta.

```
@app.get("/background/{email}")
async def background(email: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(write_notification, email, message="some notification")
    return {"message": "Notification sent in the background" }
```





¿Cuándo no es buena idea?

- App estable escrita en otro framework
  - No necesita nuevas características o no hace uso de ellas
  - No es una API
-

Gracias