FastAPI

FastAPI framework, high performance, easy to learn, fast to code, ready for production

Documentation: https://fastapi.tiangolo.com

Source Code: https://github.com/fastapi/fastapi

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.

The key features are:

- **Fast**: Very high performance, on par with **NodeJS** and **Go** (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- **Fast to code**: Increase the speed to develop features by about 200% to 300%. *
- **Fewer bugs**: Reduce about 40% of human (developer) induced errors. *
- **Intuitive**: Great editor support. Completion everywhere. Less time debugging.
- **Easy**: Designed to be easy to use and learn. Less time reading docs.
- **Short**: Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- **Robust**: Get production-ready code. With automatic interactive documentation.
- **Standards-based**: Based on (and fully compatible with) the open standards for APIs: OpenAPI (previously known as Swagger) and JSON Schema.

* estimation based on tests on an internal development team, building production applications.

Opinions

Kabir Khan - **Microsoft** (ref)

Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala - **Uber** (ref)

Kevin Glisson, Marc Vilanova, Forest Monsen - **Netflix** (ref)

Brian Okken - **Python Bytes podcast host** (ref)

Timothy Crosley - **Hug creator** (ref)

Ines Montani - Matthew Honnibal - **Explosion AI founders - spaCy creators** (ref) - (ref)

Deon Pillsbury - **Cisco** (ref)

Typer, the FastAPI of CLIs

Requirements

- Starlette for the web parts.
- Pydantic for the data parts.

Installation

```
$ pip install "fastapi[standard]"

---> 100%
```

Example

Create it

- Create a file `main.py` with:

```
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

▸ Or use `async def` ...

If your code uses `async` / `await` , use `async def` :

```
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
async def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
async def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

Note:

If you don't know, check the *"In a hurry?"* section about `async` and `await` in the docs.

Run it

```
$ fastapi dev main.py

 ╭────────── FastAPI CLI - Development mode ──────────╮
 │                                                    │
 │  Serving at: http://127.0.0.1:8000                 │
 │                                                    │
 │  API docs: http://127.0.0.1:8000/docs              │
 │                                                    │
 │  Running in development mode, for production use:  │
 │                                                    │
 │  fastapi run                                       │
 │                                                    │
 ╰────────────────────────────────────────────────────╯

INFO:     Will watch for changes in these directories: ['/home/user/code/awesomeapp']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [2248755] using WatchFiles
INFO:     Started server process [2248757]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

▸ About the command `fastapi dev main.py` ...

The command `fastapi dev` reads your `main.py` file, detects the **FastAPI** app in it, and starts a server using Uvicorn.

By default, `fastapi dev` will start with auto-reload enabled for local development.

You can read more about it in the FastAPI CLI docs.

Check it

```
{"item_id": 5, "q": "somequery"}
```

- Receives HTTP requests in the *paths* `/` and `/items/{item_id}`.
- Both *paths* take `GET` *operations* (also known as HTTP *methods*).
- The *path* `/items/{item_id}` has a *path parameter* `item_id` that should be an `int`.
- The *path* `/items/{item_id}` has an optional `str` *query parameter* `q`.

Interactive API docs

Alternative API docs

Example upgrade

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    price: float
    is_offer: Union[bool, None] = None


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}


@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {"item_name": item.name, "item_id": item_id}
```

Interactive API docs upgrade

- The interactive API documentation will be automatically updated, including the new body:
- Click on the button "Try it out", it allows you to fill the parameters and directly interact with the API:
- Then click on the "Execute" button, the user interface will communicate with your API, send the parameters, get the results and show them on the screen:

Alternative API docs upgrade

- The alternative documentation will also reflect the new query parameter and body:

Recap

```
item_id: int
```

```
item: Item
```

- Editor support, including:
  - Completion.
  - Type checks.
- Validation of data:
  - Automatic and clear errors when the data is invalid.
  - Validation even for deeply nested JSON objects.
- Conversion of input data: coming from the network to Python data and types. Reading from:
  - JSON.
  - Path parameters.
  - Query parameters.
  - Cookies.
  - Headers.
  - Forms.
  - Files.
- Conversion of output data: converting from Python data and types to network data (as JSON):
  - Convert Python types (`str` , `int` , `float` , `bool` , `list` , etc).
  - `datetime` objects.
  - `UUID` objects.
  - Database models.
  - ...and many more.
- Automatic interactive API documentation, including 2 alternative user interfaces:
  - Swagger UI.
  - ReDoc.

- Validate that there is an `item_id` in the path for `GET` and `PUT` requests.
- Validate that the `item_id` is of type `int` for `GET` and `PUT` requests.
  - If it is not, the client will see a useful, clear error.
- Check if there is an optional query parameter named `q` (as in `http://127.0.0.1:8000/items/foo?q=somequery`) for `GET` requests.
  - As the `q` parameter is declared with `= None` , it is optional.
  - Without the `None` it would be required (as is the body in the case with `PUT`).
- For `PUT` requests to `/items/{item_id}` , read the body as JSON:
  - Check that it has a required attribute `name` that should be a `str` .
  - Check that it has a required attribute `price` that has to be a `float` .
  - Check that it has an optional attribute `is_offer` , that should be a `bool` , if present.
  - All this would also work for deeply nested JSON objects.
- Convert from and to JSON automatically.
- Document everything with OpenAPI, that can be used by:
  - Interactive documentation systems.
  - Automatic client code generation systems, for many languages.
- Provide 2 interactive documentation web interfaces directly.

```
    return {"item_name": item.name, "item_id": item_id}
```

```
        ... "item_name": item.name ...
```

```
        ... "item_price": item.price ...
```

- Declaration of **parameters** from other different places as: **headers**, **cookies**, **form fields** and **files**.
- How to set **validation constraints** as `maximum_length` or `regex` .
- A very powerful and easy to use Dependency Injection system.
- Security and authentication, including support for **OAuth2** with **JWT tokens** and **HTTP Basic** auth.
- More advanced (but equally easy) techniques for declaring **deeply nested JSON models** (thanks to Pydantic).
- **GraphQL** integration with Strawberry and other libraries.

- Many extra features (thanks to Starlette) as:
  - WebSockets
  - extremely easy tests based on HTTPX and `pytest`
  - CORS
  - Cookie Sessions
  - ...and more.

Performance

Dependencies

`standard` Dependencies

- `email-validator` - for email validation.
- `httpx` - Required if you want to use the `TestClient`.
- `jinja2` - Required if you want to use the default template configuration.
- `python-multipart` - Required if you want to support form "parsing", with `request.form()`.
- `uvicorn` - for the server that loads and serves your application. This includes `uvicorn[standard]`, which includes some dependencies (e.g. `uvloop`) needed for high performance serving.
- `fastapi-cli` - to provide the `fastapi` command.

Without `standard` Dependencies

Additional Optional Dependencies

- `pydantic-settings` - for settings management.
- `pydantic-extra-types` - for extra types to be used with Pydantic.
- `orjson` - Required if you want to use `ORJSONResponse`.
- `ujson` - Required if you want to use `UJSONResponse`.

License