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](https://fastapi.tiangolo.com/#performance).
* **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](https://github.com/OAI/OpenAPI-Specification" \t "_blank) (previously known as Swagger) and [JSON Schema](https://json-schema.org/).

FastAPI stands on the shoulders of giants:

* [Starlette](https://www.starlette.io/) for the web parts.
* [Pydantic](https://docs.pydantic.dev/) for the data parts.

Dependencies[¶](https://fastapi.tiangolo.com/#dependencies)

FastAPI depends on Pydantic and Starlette.

standard Dependencies[¶](https://fastapi.tiangolo.com/#standard-dependencies)

When you install FastAPI with pip install "fastapi[standard]" it comes the standard group of optional dependencies:

Used by Pydantic:

* [email-validator](https://github.com/JoshData/python-email-validator) - for email validation.

Used by Starlette:

* [httpx](https://www.python-httpx.org/) - Required if you want to use the TestClient.
* [jinja2](https://jinja.palletsprojects.com/) - Required if you want to use the default template configuration.
* [python-multipart](https://github.com/Kludex/python-multipart) - Required if you want to support form "parsing", with request.form().

Used by FastAPI / Starlette:

* [uvicorn](https://www.uvicorn.org/) - 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.

Additional Optional Dependencies[¶](https://fastapi.tiangolo.com/#additional-optional-dependencies)

There are some additional dependencies you might want to install.

Additional optional Pydantic dependencies:

* [pydantic-settings](https://docs.pydantic.dev/latest/usage/pydantic_settings/) - for settings management.
* [pydantic-extra-types](https://docs.pydantic.dev/latest/usage/types/extra_types/extra_types/) - for extra types to be used with Pydantic.

Additional optional FastAPI dependencies:

* [orjson](https://github.com/ijl/orjson) - Required if you want to use ORJSONResponse.
* [ujson](https://github.com/esnme/ultrajson) - Required if you want to use UJSONResponse.

Pydantic models[¶](https://fastapi.tiangolo.com/python-types/#pydantic-models)

[Pydantic](https://docs.pydantic.dev/) is a Python library to perform data validation.

You declare the "shape" of the data as classes with attributes.

And each attribute has a type.

Then you create an instance of that class with some values and it will validate the values, convert them to the appropriate type (if that's the case) and give you an object with all the data.

And you get all the editor support with that resulting object.

**Path**[**¶**](https://fastapi.tiangolo.com/tutorial/first-steps/#path)

"Path" here refers to the last part of the URL starting from the first /.

So, in a URL like:

https://example.com/items/foo

...the path would be:

/items/foo

**Operation**[**¶**](https://fastapi.tiangolo.com/tutorial/first-steps/#operation)

"Operation" here refers to one of the HTTP "methods".

One of:

* POST
* GET
* PUT
* DELETE

...and the more exotic ones:

* OPTIONS
* HEAD
* PATCH
* TRACE

So, in OpenAPI, each of the HTTP methods is called an "operation".

We are going to call them "**operations**" too

You can also use the other operations:

* @app.get()
* @app.post()
* @app.put()
* @app.delete()

And the more exotic ones:

* @app.options()
* @app.head()
* @app.patch()
* @app.trace()

from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item\_id}")

async def read\_item(item\_id: int):

return {"item\_id": item\_id}

Order matters[¶](https://fastapi.tiangolo.com/tutorial/path-params/#order-matters)

When creating *path operations*, you can find situations where you have a fixed path.

Like /users/me, let's say that it's to get data about the current user.

And then you can also have a path /users/{user\_id} to get data about a specific user by some user ID.

Because *path operations* are evaluated in order, you need to make sure that the path for /users/me is declared before the one for /users/{user\_id}:

[Python 3.8+](https://fastapi.tiangolo.com/tutorial/path-params/#__tabbed_3_1)

from fastapi import FastAPI

app = FastAPI()

@app.get("/users/me")

async def read\_user\_me():

return {"user\_id": "the current user"}

@app.get("/users/{user\_id}")

async def read\_user(user\_id: str):

return {"user\_id": user\_id}

Otherwise, the path for /users/{user\_id} would match also for /users/me, "thinking" that it's receiving a parameter user\_id with a value of "me".

Predefined values[¶](https://fastapi.tiangolo.com/tutorial/path-params/#predefined-values)

If you have a *path operation* that receives a *path parameter*, but you want the possible valid *path parameter* values to be predefined, you can use a standard Python Enum.

Create an Enum class[¶](https://fastapi.tiangolo.com/tutorial/path-params/#create-an-enum-class)

Import Enum and create a sub-class that inherits from str and from Enum.

By inheriting from str the API docs will be able to know that the values must be of type string and will be able to render correctly.

Then create class attributes with fixed values, which will be the available valid values:

[Python 3.8+](https://fastapi.tiangolo.com/tutorial/path-params/#__tabbed_5_1)

from enum import Enum

from fastapi import FastAPI

class ModelName(str, Enum):

alexnet = "alexnet"

resnet = "resnet"

lenet = "lenet"

app = FastAPI()

@app.get("/models/{model\_name}")

async def get\_model(model\_name: ModelName):

if model\_name is ModelName.alexnet:

return {"model\_name": model\_name, "message": "Deep Learning FTW!"}

if model\_name.value == "lenet":

return {"model\_name": model\_name, "message": "LeCNN all the images"}

return {"model\_name": model\_name, "message": "Have some residuals"}

Path convertor[¶](https://fastapi.tiangolo.com/tutorial/path-params/#path-convertor)

Using an option directly from Starlette you can declare a *path parameter* containing a *path* using a URL like:

/files/{file\_path:path}

In this case, the name of the parameter is file\_path, and the last part, :path, tells it that the parameter should match any *path*.

So, you can use it with:

[Python 3.8+](https://fastapi.tiangolo.com/tutorial/path-params/#__tabbed_10_1)

from fastapi import FastAPI

app = FastAPI()

@app.get("/files/{file\_path:path}")

async def read\_file(file\_path: str):

return {"file\_path": file\_path}

Query Parameters[¶](https://fastapi.tiangolo.com/tutorial/query-params/#query-parameters)

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters.

[Python 3.8+](https://fastapi.tiangolo.com/tutorial/query-params/#__tabbed_1_1)

from fastapi import FastAPI

app = FastAPI()

fake\_items\_db = [{"item\_name": "Foo"}, {"item\_name": "Bar"}, {"item\_name": "Baz"}]

@app.get("/items/")

async def read\_item(skip: int = 0, limit: int = 10):

return fake\_items\_db[skip : skip + limit]

The query is the set of key-value pairs that go after the ? in a URL, separated by & characters.

For example, in the URL:

http://127.0.0.1:8000/items/?skip=0&limit=10

...the query parameters are:

* skip: with a value of 0
* limit: with a value of 10

Request Body[¶](https://fastapi.tiangolo.com/tutorial/body/#request-body)

When you need to send data from a client (let's say, a browser) to your API, you send it as a **request body**.

Import Pydantic's BaseModel[¶](https://fastapi.tiangolo.com/tutorial/body/" \l "import-pydantics-basemodel" \o "Permanent link)

First, you need to import BaseModel from pydantic:

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/body/#__tabbed_1_1)

from fastapi import FastAPI

from pydantic import BaseModel

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

app = FastAPI()

@app.post("/items/")

async def create\_item(item: Item):

return item

Path Parameters and Numeric Validations[¶](https://fastapi.tiangolo.com/tutorial/path-params-numeric-validations/#path-parameters-and-numeric-validations)

In the same way that you can declare more validations and metadata for query parameters with Query, you can declare the same type of validations and metadata for path parameters with Path.

from typing import Annotated

from fastapi import FastAPI, Path, Query

app = FastAPI()

@app.get("/items/{item\_id}")

async def read\_items(

\*,

item\_id: Annotated[int, Path(title="The ID of the item to get", ge=0, le=1000)],

q: str,

size: Annotated[float, Query(gt=0, lt=10.5)],

):

results = {"item\_id": item\_id}

if q:

results.update({"q": q})

if size:

results.update({"size": size})

return results

Query Parameters with a Pydantic Model[¶](https://fastapi.tiangolo.com/tutorial/query-param-models/#query-parameters-with-a-pydantic-model)

Declare the **query parameters** that you need in a **Pydantic model**, and then declare the parameter as Query:

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/query-param-models/#__tabbed_1_1)

from typing import Annotated, Literal

from fastapi import FastAPI, Query

from pydantic import BaseModel, Field

app = FastAPI()

class FilterParams(BaseModel):

model\_config = {"extra": "forbid"} # you might want to restrict the query parameters that you want to receive.

limit: int = Field(100, gt=0, le=100)

offset: int = Field(0, ge=0)

order\_by: Literal["created\_at", "updated\_at"] = "created\_at"

tags: list[str] = []

@app.get("/items/")

async def read\_items(filter\_query: Annotated[FilterParams, Query()]):

return filter\_query

Singular values in body[¶](https://fastapi.tiangolo.com/tutorial/body-multiple-params/#singular-values-in-body)

The same way there is a Query and Path to define extra data for query and path parameters, **FastAPI** provides an equivalent Body.

For example, extending the previous model, you could decide that you want to have another key importance in the same body, besides the item and user.

If you declare it as is, because it is a singular value, **FastAPI** will assume that it is a query parameter.

But you can instruct **FastAPI** to treat it as another body key using Body:

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/body-multiple-params/#__tabbed_5_1)

from typing import Annotated

from fastapi import Body, FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

class User(BaseModel):

username: str

full\_name: str | None = None

@app.put("/items/{item\_id}")

async def update\_item(

item\_id: int, item: Item, user: User, importance: Annotated[int, Body()]

):

results = {"item\_id": item\_id, "item": item, "user": user, "importance": importance}

return results

🤓 Other versions and variants

In this case, **FastAPI** will expect a body like:

{

"item": {

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2

},

"user": {

"username": "dave",

"full\_name": "Dave Grohl"

},

"importance": 5

}

Again, it will convert the data types, validate, document, etc.

Body - Fields[¶](https://fastapi.tiangolo.com/tutorial/body-fields/#body-fields)

The same way you can declare additional validation and metadata in *path operation function* parameters with Query, Path and Body, you can declare validation and metadata inside of Pydantic models using Pydantic's Field.

Import Field[¶](https://fastapi.tiangolo.com/tutorial/body-fields/#import-field)

First, you have to import it:

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/body-fields/#__tabbed_1_1)

from typing import Annotated

from fastapi import Body, FastAPI

from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = Field(

default=None, title="The description of the item", max\_length=300

)

price: float = Field(gt=0, description="The price must be greater than zero")

tax: float | None = None

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Annotated[Item, Body(embed=True)]):

results = {"item\_id": item\_id, "item": item}

return results

Declare Request Example Data[¶](https://fastapi.tiangolo.com/tutorial/schema-extra-example/#declare-request-example-data)

You can declare examples of the data your app can receive.

Here are several ways to do it.

Extra JSON Schema data in Pydantic models[¶](https://fastapi.tiangolo.com/tutorial/schema-extra-example/#extra-json-schema-data-in-pydantic-models)

You can declare examples for a Pydantic model that will be added to the generated JSON Schema.

[Pydantic v2](https://fastapi.tiangolo.com/tutorial/schema-extra-example/#__tabbed_1_1)[Pydantic v1](https://fastapi.tiangolo.com/tutorial/schema-extra-example/#__tabbed_1_2)

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/schema-extra-example/#__tabbed_2_1)

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

model\_config = {

"json\_schema\_extra": {

"examples": [

{

"name": "Foo",

"description": "A very nice Item",

"price": 35.4,

"tax": 3.2,

}

]

}

}

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Item):

results = {"item\_id": item\_id, "item": item}

return results

Cookie Parameters[¶](https://fastapi.tiangolo.com/tutorial/cookie-params/#cookie-parameters)

You can define Cookie parameters the same way you define Query and Path parameters.

Import Cookie[¶](https://fastapi.tiangolo.com/tutorial/cookie-params/#import-cookie)

First import Cookie:

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/cookie-params/#__tabbed_1_1)

from typing import Annotated

from fastapi import Cookie, FastAPI

app = FastAPI()

@app.get("/items/")

async def read\_items(ads\_id: Annotated[str | None, Cookie()] = None):

return {"ads\_id": ads\_id}

Header Parameters[¶](https://fastapi.tiangolo.com/tutorial/header-params/#header-parameters)

You can define Header parameters the same way you define Query, Path and Cookie parameters.

Import Header[¶](https://fastapi.tiangolo.com/tutorial/header-params/#import-header)

First import Header:

[Python 3.10+](https://fastapi.tiangolo.com/tutorial/header-params/#__tabbed_1_1)

from typing import Annotated

from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")

async def read\_items(user\_agent: Annotated[str | None, Header()] = None):

return {"User-Agent": user\_agent}

response\_model Priority[¶](https://fastapi.tiangolo.com/tutorial/response-model/#response_model-priority)

If you declare both a return type and a response\_model, the response\_model will take priority and be used by FastAPI.

Import Form[¶](https://fastapi.tiangolo.com/tutorial/request-forms/#import-form)

pip install python-multipart

Import Form from fastapi:

[Python 3.9+](https://fastapi.tiangolo.com/tutorial/request-forms/#__tabbed_1_1)

from typing import Annotated

from fastapi import FastAPI, Form

app = FastAPI()

@app.post("/login/")

async def login(username: Annotated[str, Form()], password: Annotated[str, Form()]):

return {"username": username}