**Introduction to FastAPI**



Photo by [Jordan Harrison](https://unsplash.com/@jordanharrison?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

Some of its main 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 productivity by using Python type hints without changing the code. Static type checkers like mypy improve editor support.

• Easy: Great editor support. Completion everywhere. Less time debugging.

• Intuitive: Easy to learn, fast to code, intuitive to read.

• Robust: Get production-ready code. With automatic interactive documentation.

• Standard: Follows (and helps evolve) the Open API Specification (formerly Swagger), OAuth2, JWT, etc.

• Typycal: Uses Python type hints for validation, serialization and interactive documentation.

• Interactive Documentation: Beautiful interactive documentation for your API with Swagger UI integrated.

• OAuth2: Support for OAuth2 password, server to server, single sign-on etc. Integrated with fastapi-users.

• Validation: Using Pydantic to validate request data, query parameters, headers, cookies, etc.

To install FastAPI, run the following pip command:

pip install fastapi

This will install FastAPI along with its dependencies like Starlette, Pydantic, etc.

**Your First API**

Here is a simple “Hello World” API with FastAPI:

from fastapi import FastAPI  
  
app = FastAPI()  
  
@app.get("/")  
def read\_root():  
 return {"Hello": "World"}

This API has one endpoint, /, that returns a JSON response {"Hello": "World"}.

To run this API, use the ASGI server Uvicorn:

uvicorn main:app --reload

Uvicorn will run the API on http://localhost:8000. The --reload flag reloads the server automatically on code changes.

**Path Parameters**

Path parameters are parameters in the path of a URL. They are defined using braces {}. For example, an API with a path parameter for a person's name:

@app.get("/greet/{name}")   
def greet(name: str):   
 return f"Hello {name}"

When you call /greet/John, it will translate to:

name = "John"

And the response will be: Hello John

The type of the path parameter is declared using standard Python type hints and FastAPI uses Pydantic to convert the path parameter to that type.

**Query Parameters**

Query parameters are key-value pairs in the query string of a URL. For example:

/items?category=clothes&brand=Zara

Here category and brand are query parameters.

You can declare query parameters in FastAPI as follows:

@app.get("/items/")  
def read\_items(category: str, brand: str):   
 ...

Now when you call /items?category=clothes&brand=Zara, FastAPI will validate the parameters and pass them to the function:

category = "clothes"   
brand = "Zara"

The types of the query parameters are, again, defined using Python type hints. FastAPI uses Pydantic to validate the parameters and serialize them to the declared types.

**Request Body**

The request body contains the data in a request, for example, in a POST request. You can declare a request body in FastAPI as follows:

from pydantic import BaseModel  
  
class Item(BaseModel):  
 name: str  
 description: str  
 price: float  
  
@app.post("/items/")  
def create\_item(item: Item):  
 ...

Here we have defined an Item model using Pydantic. In the path operation, we use that model to validate and serialize the request body. When you call /items/ with a request body like:

{  
 "name": "Foo",  
 "description": "A new item",  
 "price": 45.2   
}

FastAPI will validate it using the Item model and pass the data to your function:

item = Item(name="Foo", description="A new item", price=45.2)

**Response Model**

You can also declare a model for your API response, and FastAPI will automatically convert it to JSON:

@app.get("/items/")  
def read\_items():  
 items = [  
 { "name": "Foo", "description": "A new item", "price": 45.2 },  
 { "name": "Bar", "description": "Another item", "price": 10.5 }  
 ]  
 return items   
  
# Response:   
[  
 {  
 "name": "Foo",  
 "description": "A new item",  
 "price": 45.2  
 },  
 {  
 "name": "Bar",   
 "description": "Another item",  
 "price": 10.5   
 }  
]

But a better way would be to define an Item Pydantic model and return a list of instances of it:

from pydantic import BaseModel  
  
class Item(BaseModel):  
 name: str  
 description: str   
 price: float  
  
@app.get("/items/")  
def read\_items():  
 items = [Item(name="Foo", description="A new item", price=45.2),   
 Item(name="Bar", description="Another item", price=10.5)]  
 return items  
  
# Response:  
[  
 {  
 "name": "Foo",  
 "description": "A new item",  
 "price": 45.2  
 },  
 {  
 "name": "Bar",   
 "description": "Another item",  
 "price": 10.5   
 }   
]

Now FastAPI will automatically convert the Item objects to JSON for the response.

**Data Manipulation**

You can access data from requests in your API endpoints using normal Python function arguments. For example:

@app.post("/items/")  
def create\_item(item: Item):   
 database.add\_item(item) # Add to some database

Here you can access the item data from the request body and use it to add an item to a database.

You can also inject dependencies using FastAPI’s Dependency functionality. For example, to inject a database connector:

database = Database() # A database connector  
  
@app.post("/items/")  
def create\_item(item: Item, database=Depends(database\_connector)):   
 database.add\_item(item)

Now the database will be injected as a dependency in the path operation.

FastAPI also has integration with SQLAlchemy for easier database handling.

**Authentication**

FastAPI supports OAuth2 out of the box and also has extensions to simplify auth flows.

You can implement basic auth as follows:

@app.get("/protected")  
def proteceted(password: str, required\_password="secret"):  
 if password == required\_password:  
 return "Success!"  
 return "Invalid password"

For token authentication, you can check for the Authorization header:

@app.get("/protected")  
def protected(authorization: str):  
 if authorization == "token 12345":  
 return "Success!"  
 return "Invalid token"

For OAuth2, FastAPI integrates with FastAPI Users which supports:

* OAuth2 password flow
* OAuth2 client credentials flow
* OpenID Connect authentication
* JWT tokens
* Much more

You can install it with pip install fastapi-users and check out the docs to integrate OAuth2 easily in your FastAPI app.

**Testing**

FastAPI has test client functionality built-in thanks to Starlette. You can test your API as follows:

from fastapi.testclient import TestClient  
  
client = TestClient(app)  
  
def test\_read\_root():  
 response = client.get("/")  
 assert response.status\_code == 200  
 assert response.json() == {"Hello": "World"}

This will call your API endpoint and verify the response.

You can test path parameters, query parameters, request bodies, auth etc. For example:

def test\_greet():  
 response = client.get("/greet/John")   
 assert response.status\_code == 200  
 assert response.text == "Hello John"  
  
def test\_create\_item():  
 response = client.post("/items/", json={"name": "Foo", "description": "A new item"})  
 assert response.status\_code == 200

To test OAuth2 flows, you can call the OAuth2 endpoints directly