

API → Communication Contract between a **client** and a **server**
HTTP → Defines how **communication** happens





FastAPI

- Accepts HTTP requests
- Understand HTTP rules
- Validate input
- Call Backend Logic
- Return HTTP responses

Real World Analogy

Restaurant



- 1. Customer → Client
- 2. Menu → API
- 3. Waiter → API Server (FastAPI)
- 4. Kitchen → Backend Logic (Server)
- 5. Food → Response

What is FastAPI ?

API Server | Open-source Python Framework for creating web APIs
Build on Starlette

- Modern Python features like Type Hints. → `name: str`
- Build on Starlette → Runs on ASGI (Asynchronous Server Gateway Interface)
- Uses Pydantic → for data validation

Why FastAPI ?

Its is blasting fast – one of the fastest python frameworks

- Very fast on par with Node.js or Go
- Fewer Bugs
- Very less Code

→ **NETFLIX** **Uber**  Microsoft

FastAPI Capabilities

- Maps URLs and HTTP methods directly to Python functions

GET /users?

- Path, Query, Headers, Cookies or JSON bodies
- Automatically convert the value returned by your path operation function to a response
 - JSON response as default
 - Also return HTML, Plain text or file streams

Automatic Superpowers

- Automatic Data validation
- Interactive docs
 - /docs
 - /redoc
- Authorization and Database connections.
- Clean and Scalable.



FastAPI

- Accepts HTTP requests
- Understand HTTP rules
- Validate input
- Call Backend Logic
- Return HTTP responses

Next |  Hello World to FastAPI



Your first FastAPI code



Hello World!

main.py

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

Creates Application Object

```
@app.get("/")
```

GET request

```
async def home():
```

```
    return {"message": "Hello World"}
```

Return JSON response

```
>>> uvicorn main:app --reload
```

Filename

Application Object variable name



Path

API endpoint, create it using decorators

→ Path give you complete **control over how your API routes requests**. They are building blocks of scalable, maintainable, and reliable APIs.

main.py

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/items")
```

```
async def get_items():
```

```
    return {"items": ["item 1",  
                      "item 2",  
                      "item 3"]
```

```
}
```

Path/URL path

http GET <http://127.0.0.1:8000/items>

Response:

```
{"items": ["item 1", "item 2", "item 3"]}
```

⚡ Path

API endpoint, create it using decorators

→ These kind of path are **dynamic variable path**. Where variables embedding directly into the URL.

main.py

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/items/{item_id}")
```

```
async def get_items(item_id: int):
```

```
    return {"item_id": item_id}
```

Dynamic Path/URL path

Pydantic Data Validation

```
http GET http://127.0.0.1:8000/items/42
```

Response:

```
{"item_id": 42}
```

```
http GET http://127.0.0.1:8000/items/abc
```

Response: **INVALID Path Parameter**



Route order is Important

✓ Correct Order

main.py

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/items/me")
async def get_me():
    return {"items": "me"}
```

```
@app.get("/items/{item_id}")
async def get_items(item_id: int):
    return {"item_id": item_id}
```

http GET <http://127.0.0.1:8000/items/42>

Response:

```
{"item_id": 42}
```

✗ Incorrect Order

main.py

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/items/{item_id}")
async def get_items(item_id: int):
    return {"item_id": item_id}
```

```
@app.get("/items/me")
async def get_me():
    return {"items": "me"}
```

http GET <http://127.0.0.1:8000/items/me>

Response: **Invalid Request**



APIRoute

FastAPI Python framework used to structure, organize, and manage API routes (path operations) in a modular way

APIRouter is used to:

- Organize routes into multiple files
- Build modular applications
- Improve scalability
- Follow production-level architecture

Instead of putting all routes in main.py, we divide them into logical modules like:

- users.py
- products.py
- orders.py

APIRoute

FastAPI Python framework used to structure, organize, and manage API routes (path operations) in a modular way

Folder Structure

```
project/  
├── main.py  
└── routers/  
    ├── users.py  
    └── products.py
```

main.py

```
from fastapi import FastAPI, APIRouter  
from routers import products  
  
app = FastAPI()  
app.include_router(products.router, prefix="/products", tags=["Products"])
```

products.py

```
from fastapi import APIRouter  
  
router = APIRouter()  
  
@router.get("/products")  
async def get_products():  
    return {"message": "Products"}
```



Query Operations

Performing data filter process



Automatic Documentation

Swagger UI

Automatic Documentation

FastAPI automatically generates automatic documentation without writing extra code 🤖

- Interactive API documentation
- Request & response schema
- Validation details
- Example testing interface

Swagger UI → Interactive Documentation

- By default, FastAPI provides documentation using **Swagger UI**.

```
http://127.0.0.1:8000/docs
```



Templates

Rendering HTML

Templates

Rendering HTML Pages with FastAPI

- Return HTML pages instead of JSON
- Build forms, dashboards, login pages etc
- Render dynamic data inside HTML

Template Engine → Jinja2

```
pip install jinja2  
      (or)  
poetry add jinja2
```


Templates

folder structure and import metionds

Folder Structure

```
project/  
├── main.py  
└── templates/  
    ├── index.html  
    └── forms.html
```

main.py

```
from fastapi import FastAPI, Request  
from fastapi.response import HTMLResponse  
from fastapi.templating import Jinja2Templates  
  
app = FastAPI()  
  
templates = Jinja2Templates(directory = "templates")
```

main.py

```
@app.get("/", response_class = HTMLResponse)  
async def home(request: Request):  
    users_data = ["User1", "User2", "User3"]  
  
    return templates.TemplateResponse(  
        "index.html",  
        {  
            "request": request,  
            "users": users_data  
        }  
    )
```



{{ Variables }}

Insert Variables in HTML

Insert Variables

Rendering HTML Pages with FastAPI

```
return templates.TemplateResponse(
    "index.html",
    { "request": request,
      "name": "John"
    } )
```

Feature	Syntax	Example
Insert Variable	<code>{{ variable_name }}</code>	<code>{{ name }}</code> _____
If Condition	<code>{% if condition %}</code> ... <code>{% endif %}</code>	<code>{% if age >= 18 %}</code> Adult <code>{% endif %}</code>
If-Else	<code>{% if condition %}</code> ... <code>{% else %}</code> ... <code>{% endif %}</code>	<code>{% if age >= 18 %}</code> Adult <code>{% else %}</code> Minor <code>{% endif %}</code>
If-Elif-Else	<code>{% if %}</code> ... <code>{% elif %}</code> ... <code>{% else %}</code> ... <code>{% endif %}</code>	<code>{% if marks >= 90 %}</code> A <code>{% elif marks >= 75 %}</code> B <code>{% else %}</code> C <code>{% endif %}</code>
For Loop	<code>{% for item in items %}</code> ... <code>{% endfor %}</code>	<code>{% for user in users %}</code> <code>{{ user }}</code> <code>{% endfor %}</code>
Loop Index	<code>{{ loop.index }}</code>	<code>{{ loop.index }}</code> - <code>{{ user }}</code>



Base Template

Template Inheritance

Base Template

Template Inheritance

Layout Template

base.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      {% block title %}
        My App
      {% endblock %}
    </title>
  </head>
  <body>
    <header>
      <h1>My website</h1>
    </header>
    <hr>
    {% block content %}
    {% endblock %}
    <hr>

    <footer>
      <p>Copyright 2026</p>
    </footer>
  </body>
</html>
```

Child Template

home.html

```
{% extends "base.html" %}
{% block title %}Home Page{% endblock %}

{% block content %}
  <h2>Welcome to Home Page</h2>
  <p>This content is dynamic.</p>
{% endblock %}
```



Form

POST request | Receive Form Data



POST request

< > ↻ ⓘ 127.0.0.1:8000/users/form

Enter Name Enter Age Submit



< > ↻ ⓘ 127.0.0.1:8000/users/submit

Pretty-print ☒

```
{
  "username": "John",
  "age": 28
}
```



Upload Image

Template Inheritance



MongoDB in FastAPI

Integrating FastAPI with NoSQL Database

Query Operations

FastAPI Python framework used to structure, organize, and manage API routes (path operations) in a modular way

APIRouter is used to:

- Organize routes into multiple files
- Build modular applications
- Improve scalability
- Follow production-level architecture

Instead of putting all routes in main.py, we divide them into logical modules like:

- users.py
- products.py
- orders.py